Department of Computer Science

Technical Report

Undergraduate Dissertation: An Investigation of Techniques for Improving the Performance of a Pittsburgh Approach Learning Classifier System

Anne Stacey

Technical Report 2004-09 **May 2004**

ISSN 1740-9497

Contact Address:

Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom

URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

An Investigation of Techniques for Improving the Performance of a Pittsburgh Approach Learning Classifier System

Anne Stacey

BSc (Hons) Computer Science

University of Bath

May 2004

Abstract

Learning Classifier Systems (Holland 1978) are a Machine Learning technique in which a set of simplified production rules are discovered to solve a given problem. The system must identify patterns within the data it receives from the problem environment, enabling it to classify other, previously unseen inputs. The rules, known as classifiers, are selected and manipulated using a Genetic Algorithm (Holland 1975). This dissertation focuses on Pittsburgh approach classifier systems (Smith 1980), where many candidate rule sets compete with one another. Such systems tend to be slow since the Genetic Algorithm must work with large structures, namely entire sets of classifiers; increasing their efficiency is therefore an important research topic. We propose a technique in which the rule sets are compressed before manipulation and then re-expanded using principles from Grammatical Evolution. We then go on to explore methods of controlling bloat, a problem where rule sets grow out of control, slowing down the system and reducing the effectiveness of the search. A new algorithm is implemented, in which weak classifiers are identified and explicitly removed from the candidate solutions. This results in a considerable improvement to the system's performance on a number of Data Mining tasks.

Acknowledgements

With thanks to my supervisor, Dr Alwyn Barry, for his many helpful suggestions and encouragement throughout the project. I also wish to thank both of my parents for their ongoing support.

AN INVESTIGATION OF TECHNIQUES FOR IMPROVING THE PERFORMANCE OF A PITTSBURGH APPROACH LEARNING CLASSIFIER SYSTEM

Submitted by Anne Stacey

for the degree of BSc (Hons) Computer Science

of the University of Bath

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Contents

Chapter 1

Introduction

Learning Classifier Systems are a branch of Machine Learning, a field concerned with creating computer systems that have the ability to 'learn' certain tasks and improve their own performance over time. For a given problem, an LCS aims to find a set of production rules called classifiers which describe how to solve that problem. The left-hand side of each rule is a 'condition' string against which inputs from the environment can be matched, while the right-hand side predicts the class of any data instances matching the rule, i.e. it suggests an appropriate action to be taken when those conditions are met. The condition string may contain "don't care" symbols called wildcards to indicate that certain features of the environment are irrelevant when choosing an action; the process of determining where to place these wildcards is known as generalisation. The system's task, therefore, is to identify patterns within the data it receives from the problem environment, and thus be capable of accurately classifying previously unseen inputs. In other words, the system chooses appropriate actions by recognising when a situation is similar to one it has experienced before.

Most LCS implementations use a Genetic Algorithm to repeatedly refine the internal rule set until a complete and consistent description of the target concept is found. Genetic Algorithms (Holland 1975) are an efficient search technique used for problems where the solution space is large or complex; they are so named because they were inspired by genetics and the process of natural selection. Just as the characteristics of an organism are encoded on chromosomes which are then recombined during sexual reproduction, it is possible to create simple structures that encode candidate solutions to a particular machine learning or optimisation task, and simulate breeding amongst these structures. The solutions are assigned 'fitness' values according to how well they perform in the problem

environment, and selection of parents for the next generation is biased in favour of individuals with a higher fitness, so that the quality of solutions tends to improve over a number of generations. Offspring are produced via 'genetic operators' such as crossover, where sections of the two parent individuals are swapped around, and mutation, where the value of a bit is randomly altered. These operators are discussed in detail within the Literature Review (chapter 2).

Two distinct approaches have emerged for Learning Classifier Systems, each named after the university where they were first developed: the Michigan approach (Holland 1978) and the Pittsburgh approach (Smith 1980). In the former, a single population of classifiers is maintained and the GA is employed to select and manipulate the rules themselves. In the latter, each individual manipulated by the GA is a complete rule set. Pittsburgh systems overcome the problem of good classifiers being eliminated due to competing with one another, but the GA must now work with much larger structures (i.e. long strings representing the concatenation of many classifiers), which tends to make such systems extremely slow to find solutions. An important area of research, therefore, is how to increase the efficiency of Pittsburgh Learning Classifier Systems, and this will be the focus of the dissertation.

The Literature Review gives an overview of previous work in the field, describing the various styles of Learning Classifier System and the problems associated with each approach. We also propose a method of improving the efficiency of Pittsburgh systems by using principles from Grammatical Evolution, a relatively new technique developed at the University of Limerick (Ryan 1998a) for evolving expressions or programs in any language. The language to be generated is described using a BNF grammar definition, and a linear genome is used to control how the grammar definition is mapped to an actual expression or program. We suggest that the large individuals within the Pittsburgh LCS could be reduced to a more compact form before manipulation by the GA, and then re-expanded into full rule sets using Grammatical Evolution. However, this is ultimately not investigated in the current project, as explained below.

Chapters 3 and 4 discuss the requirements that were identified for an experimental Pittsburgh LCS, and the design choices that were made as a result. Chapter 5 describes a set of standard Data Mining tasks that would be used to test various aspects of the system's capabilities. In chapter 6, we discuss the problems that were encountered when testing the initial versions of the system; namely the well-known 'bloat effect' in which fitness-based selection is responsible for rule sets tending to grow out of control. This results in poor-

quality solutions which are much larger than necessary, and also slows down the search to such a degree that testing may become impractical. The remainder of the dissertation is dedicated to investigating new methods for controlling this bloat. We hypothesise that techniques from the Michigan-style classifier system XCS (Wilson 1995), including accuracy-based fitness, subsumption and deletion of inaccurate rules, may be an effective method of bloat control if transferred into a Pittsburgh LCS. Chapter 7 presents the promising initial results of this investigation, and shows that the performance of our system (known as A-PLUS) on the 6-multiplexer problem was further improved by replacing subsumed classifiers with copies of the more general rule instead of merely deleting them. In chapter 8, we experiment with removing parts of the algorithm in order to better understand the effects of each technique. Chapter 9 demonstrates the performance of A-PLUS on a range of harder problems designed to test its generalisation capabilities and its reaction to noisy data. Finally, chapter 10 summarises our findings and suggests further improvements and areas of investigation.

Chapter 2

Literature Review

2.1 Introduction

There is much debate as to the qualities a system would need to possess before it could be described as showing 'artificial intelligence'. However, it is widely felt that one important requirement of such a system is the ability to learn; that is, the system would be capable of improving its own knowledge base or performance on a task over time. There is an entire research field devoted to machine learning techniques, and a variety of approaches have been investigated, including the use of neural networks and classifier systems. The latter will be the focus of this project.

A Learning Classifier System works by evolving pattern-matching capabilities that enable it to correctly determine the class of each input received from the system's environment. The better the internal rules are, the more 'skilled' the system becomes at making sense of new and unfamiliar inputs, based on what it has already seen. The learning facility is provided by a Genetic Algorithm which is employed to selectively recombine the system's internal rules, enabling them to 'evolve' over time just as biological organisms evolve to become better suited to their environment.

There are two main approaches to the design of Learning Classifier Systems, which will be examined in a later section. The version to be investigated in this project is known as the Pittsburgh approach, in which the aim is to develop an optimal set of rules for solving a given problem. This method is currently slow to find results, due to the size of the structures on which the Genetic Algorithm must operate. Here, a possible improvement to the system is suggested, in which the rule sets are reduced to a simpler representation

before being used to breed new sets. These transformations can be performed using Grammatical Evolution, a relatively new system for generating expressions or programs via a BNF grammar definition.

The following sections will explore these various topics in more detail and explain how Grammatical Evolution could be integrated within a Pittsburgh Learning Classifier System.

2.2 Genetic Algorithms

2.2.1 Background to Genetic Algorithms

Genetic Algorithms, commonly known as GAs, are a form of search procedure loosely based on ideas drawn from natural evolution and genetics. Our planet is home to a vast range of biological organisms which have adapted to life in very different environments; Charles Darwin attributed this to 'natural selection' (Darwin 1859) leading to survival of only the fittest organisms in a given population. For example, if a particular habitat requires organisms to have a long neck in order to reach the main food supply, then individuals having long necks are more likely to survive and reproduce, while individuals lacking this trait will die off sooner. The eventual result is that genes encoding the long neck trait will be passed down through the generations, until it becomes a standard feature of organisms living in that habitat. Furthermore, when two organisms mate and produce offspring, characteristics of each parent are combined in new ways; so if each parent has a particularly strong feature that is not possessed by the other, it is possible that the offspring will inherit both features and be even better suited to the environment than its parents. Finally, every so often a small genetic mutation will occur, resulting in an individual possessing a trait that was not present in the population before. Occasionally this trait will prove beneficial in the struggle to survive, and will be passed on to other individuals through breeding.

It is clearly evident that this evolutionary process has been successful in nature, and even more significant is the timescale in which it has occurred. It is extremely impressive that such diverse life forms, each perfectly suited to a particular climate with different social pressures, have developed within a relatively short space of time. Observing this in the 1960s, John Holland at the University of Michigan was inspired to investigate whether a similar process could be implemented in computers, enabling complex problems to be

solved just as quickly and reliably. Together with his colleagues and students, he developed Genetic Algorithms, which use analogies of the genetic operators found in real biology (Holland 1975). Variations on the theme have been tried and tested over the years, but the GAs in use today remain very similar to Holland's original version.

2.2.2 How Genetic Algorithms work

In real genetics, the basic structures are lengths of DNA called chromosomes. These are made up of a number of genes, each of which encodes a particular trait such as eye colour; the range of 'values' that the gene can take are called the alleles. The particular combination of alleles on a chromosome can be thought of as a 'blueprint' for that individual; this is called the genotype. The resulting physical characteristics of the organism are known as the phenotype.

The corresponding structures manipulated by a Genetic Algorithm are designed to be a simplified model of these chromosomes; usually binary strings are used, although it is possible to use more complex structures such as strings of real numbers or trees. In a binary string, each bit may represent a particular trait which can have one of two values (e.g. on or off, blue or brown). Alternatively, the string as a whole might represent a number value, for example – the structure is flexible enough to be useful for encoding a range of problems. A particular combination of bit values can therefore be used to generate an actual solution in the given problem domain; this is a kind of genotype-to-phenotype mapping. The aim of the GA is to develop an optimal genotype by applying simple genetic operators to carefully chosen individuals (i.e. candidate solutions).

The three main natural mechanisms identified by Holland were selection, crossover and mutation. (A further operator, inversion, was also simulated, but is now less commonly used.) Selection is the process of determining which individuals will become parents for the next generation. In nature, weaker individuals may not survive long enough to reproduce, and organisms tend to choose a mate who displays desirable characteristics (e.g. bigger antlers). In other words, the probability of an individual going on to breed is in some way proportional to its 'fitness' – how well the organism is suited to its environment. Holland simulated this by assigning a fitness value to each individual string. As a simple example, if the aim were to breed a string consisting of all 1's, then the more 1's a string contained, the higher a fitness value it would be given. When strings were then being selected for breeding, the method used would favour individuals of higher fitness. There

are a number of ways this can be done, but a simple and popular method is likened to spinning a weighted roulette wheel. The higher an individual's fitness value, the larger its slot on the roulette wheel, and the more likely it is to be selected as a consequence. The probability of individual *i* being selected is given by:

$$p_i = \frac{\text{fitness}_i}{\text{sum of all fitnesses}}$$

Note that since probabilities are used, weaker individuals still have a chance of being selected to reproduce, though it is reasonable to expect that the average fitness of the population will increase over time using this method.

Assuming the selection phase is completed before the genetic operators are applied, this process effectively produces copies of certain individuals from the current population, with stronger individuals tending to receive more copies than weaker ones. Some individuals might not be copied at all. This results in a 'mating pool' biased towards individuals of higher fitness (Goldberg 1989a). Some of these individuals will be placed straight into the new population, unchanged – this can be thought of as producing an offspring that is genetically identical to its parent. However, in order to simulate the genetic recombination that occurs naturally during sexual reproduction, other individuals will be used for 'breeding' via the crossover operator. A predefined crossover probability determines how often this operator will be applied.

Crossover between two parent strings is carried out as follows. One or more cross sites are chosen at random along the length of the strings, and then a portion of the strings occurring between these points are exchanged. For example, if single-point crossover is used, then each parent string is divided into a 'head' and 'tail' section, and the tails are swapped over. If two crossover points are chosen, then the substrings between the two points are swapped. The operation results in two 'child' strings being produced, each combining traits from the two parent strings. These new strings are then evaluated to determine their fitness, before being added to the new population.

Although crossover is a fairly effective way of exploring possible solutions to the task at hand, it is not perfect. The problem is that the nature of the recombination process can lead to vital alleles being irrecoverably lost. An allele is a particular value of a gene – for example, the third bit in the string having value '1'. If this allele turns out to be important

for solving the task, then if it disappears from the population during the breeding process (or is not present to begin with), the optimal solution cannot be found. For this reason, Holland also used another operator, analogous to genetic mutation. Each position in a string has a very small probability (often 0.001) of having its value randomly altered – in a binary string, this would mean flipping a bit from 0 to 1 or vice versa. The mutation rate is, again, specified by the user before the Genetic Algorithm is run. This operator provides an opportunity for a key allele to be reintroduced, but must be applied sparingly to avoid turning the process into a random search for solutions.

Holland used an additional genetic operator called inversion, which has not been as well understood as those described above, and is often left out nowadays. The purpose of inversion is essentially to find better codings as well as better sets of alleles. Two points along the length of a chromosome are chosen, and the section between these points is 'reversed' – e.g. if this substring was 111000, it would be changed to 000111. In order for this operator to be useful, the meaning of an allele needs to be position independent, i.e. a bit value will retain its meaning even if it is moved to a different point on the chromosome in this way. This can be done by 'naming' each allele, and transferring this name information along with the bit value when inversion is applied. In that case, a string produced through the inversion operator is not a new point in the search space, but merely a different representation of the original point (Smith 1980). A similar effect happens in nature, where genes can be reordered and yet still be responsible for producing the same enzyme. Inversion helps to combat the problem which occurs when useful combinations of alleles are widely spaced out along a chromosome – the problem being that crossover is likely to disrupt this combination. When inversion is used, there is a chance that the related alleles will be moved closer together, and so the 'defining string segment' will be less vulnerable to disruption by the crossover operator.

The use of an inversion operator can potentially lead to invalid chromosomes being generated through crossover. For example, if one parent chromosome has an 'eye colour' gene located near the 'head' end, whilst the corresponding gene on the other parent has been shifted towards the 'tail' end, then crossover will result in a child string that has two eye colour genes and a child that has none at all. There are a number of possible strategies for avoiding this situation, such as temporarily reordering one string to make it 'compatible' with the other parent before crossover is applied. Smith points out that the above problem does not arise if the interpretation of a string is relatively independent of the positions of the values, for example if the string represents a set of unordered rules.

These straightforward mechanisms provide a surprisingly effective method for finding solutions to a given problem. The user simply chooses a way of encoding the problem features onto (for example) a binary string, and randomly generates an initial population of such strings, each representing a possible solution to the problem. Each individual is assigned a fitness value according to how well it solves the problem, and the operations described above are used to breed individuals together in the hope of combining useful characteristics from each. The average fitness increases with successive generations, until ultimately the best solution(s) are found. Depending on the nature of the problem and the user's preferences, the algorithm can either terminate once a 'perfect' solution is discovered, or it can be set to run for a finite number of generations, at which point the individual with the highest fitness is selected as the solution.

It should be noted that the key to a Genetic Algorithm's progress lies in selecting the right individuals for reproduction, since crossover and mutation by themselves provide little more than a random walk through the solution space. More sophisticated Genetic Algorithms have been investigated, such as those which allow a number of 'species' to emerge, each occupying a separate 'niche' in the solution – for example when it is desirable to identify multiple peaks rather than just one. It is helpful to observe that in nature, where individuals choose their own mates instead of being paired up by an external force, there is a strong tendency for organisms to select a mate from within their own species (i.e. one who exhibits similar characteristics). This process can be simulated in a GA by allowing two individuals to breed only if they are sufficiently similar, leading to the development of distinct species which may converge around different peaks. The lack of inter-species breeding means that low-performance 'hybrid' offspring are less likely to be produced.

Other extensions of the original Simple Genetic Algorithm include Messy GAs (Goldberg 1989b). These store name information as well as the value of a gene, as described for the inversion operator above. Evolution is divided into two phases: the primordial phase, where short, highly fit 'building blocks' are constructed, and the juxtapositional phase, where these building blocks are combined. Crossover is replaced by two operators called cut and splice. The algorithm is 'messy' because the chromosomes produced can be overspecified, meaning they contain multiple values for the same gene, or underspecified, meaning certain genes are missing. When evaluating the chromosomes, the former problem is solved by scanning the genes from left to right and using the first value that appears for a given gene, ignoring any further, contradictory values. For underspecified chromosomes, the missing gene values are filled with the corresponding alleles from a template.

As our understanding of real genetics continues to improve, it is likely that new and more powerful operators will be introduced to Genetic Algorithms. Research into this area includes that of Stephanie Forrest and Melanie Mitchell, who have described and analysed an 'idealized genetic algorithm' (Mitchell 1997) and are interested in studying which versions of a GA will most closely conform to this, and under what parameter settings.

2.2.3 The strength of Genetic Algorithms

Not only are GAs appealing on a theoretical level, having been based on highly successful natural processes, but they have been proven to be effective in a wide variety of domains. The primary advantage of GAs over traditional problem-solving methods is that they are 'robust', offering a combination of breadth and efficiency (Goldberg 1989a). They can be applied to many different problems because they use a kind of 'black box' approach, manipulating mere codings of the decision variables instead of being restricted by problem-specific information.

In addition to this, Genetic Algorithms are much less likely to converge on a false peak. This is a phenomenon that occurs in search methods such as hill-climbing, where a point in the space of solutions is selected randomly, and the search continues in whichever direction will lead to a better solution (imagine moving towards a higher point on a line graph). This can result in the search terminating at a local optimum which may not be the highest peak in the entire solution space. Genetic Algorithms are able to avoid this problem by searching from a population of solutions in which diversity is encouraged.

Genetic Algorithms are particularly useful whenever the solution space to a problem is far too large for an exhaustive search to be practical or even possible. The degree of randomisation within the process helps to keep decision-making free of bias in situations where there is nothing to indicate which choice might be better. GAs strike an effective balance between exploitation, i.e. reusing ideas that are proven to be good, and exploration, where previously untested ideas are tried out.

GAs have found useful applications in fields ranging from finance to criminology to the design of jet engines. This project will focus on their role in Machine Learning, and in particular, Learning Classifier Systems.

2.3 Learning Classifier Systems

2.3.1 What is a Learning Classifier System?

Learning Classifier Systems are a branch of Machine Learning which use Genetic Algorithms to learn how to solve a given problem. An LCS has a set of internal rules called classifiers, which it uses to determine the 'class' of incoming environmental signals and therefore decide upon the appropriate action to take. The Genetic Algorithm operates on these rules to produce better ones, gradually improving the system's performance, so that over time it 'learns' how to solve the problem more efficiently.

Many different Learning Classifier Systems have been implemented, and a division into two major approaches has emerged. These are known as the Michigan and Pittsburgh approaches, which will be discussed later. The first classifier system, CS-1, was designed by John Holland and Judith Reitman (Holland 1978); this was an example of a Michigan LCS. Following this, Stephen Smith developed a system called LS-1 in an attempt to improve on the original idea, by having the GA search for optimal combinations of rules, i.e. good populations, as well as good individual rules (Smith 1980). This was the first design of a Pittsburgh LCS. An example of a more recent implementation of the Pittsburgh LCS is GABIL (De Jong 1993).

GABIL is described by its creators, Kenneth De Jong and William Spears, as a 'concept learning program'. They explain that a 'concept' can be thought of as a subset of points in an *n*-dimensional feature space, and that any point within this feature space represents either a positive or negative example of the concept (De Jong 1991). The classifier system is given a description of the feature space, and then presented with a set of these examples that have already been correctly classified as positive or negative. Based on this information, the system must try to generate a 'concept description' which it can use to classify further examples. If it makes inaccurate classifications, then its rule set must be flawed, and a Genetic Algorithm is called upon to refine the rules. The aim is to find a concept description which is both consistent, meaning it does not cover (or 'recognise') any negative examples, and complete, meaning it recognises all the positive examples (Kodratoff 1988). Once the system is able to consistently achieve a 100% success rate in classifying the examples presented to it, it can be said that the system has learned the concept.

A number of important issues must be considered when designing an LCS. Firstly, a suitable concept description language is required; it must be capable of expressing complex concepts, i.e. large subsets of the feature space, in a succinct manner, while simultaneously being able to capture irregularities. GABIL uses rules; some other classifier systems use structures such as decision trees. Secondly, since there may be a large or infinite set of possible concept descriptions which are consistent with any given finite set of examples, designers generally introduce some explicit or implicit bias to their system (e.g. preference for shorter or less complex descriptions).

Thirdly, there is a choice between two main evaluation approaches, known as batch mode and incremental mode. In batch mode, the system is first presented with a set of examples called the training set, which it uses to generate and refine its concept description. After this phase, no further learning takes place, and the concept description is tested on a new set of examples; the validity of the description is given as the percentage of correct classifications made. By contrast, if incremental mode evaluation is used, the learning process never stops. The system produces a concept description based on the examples it has seen so far, and uses it to classify the next example. If it is wrong, the concept description is refined so that it is once again consistent with all the examples seen, and the process continues. The system is evaluated in terms of 'learning curves' measuring its predictive performance over a period of time. De Jong and Spears chose to use this latter approach, as they felt that the continuous model of concept learning was more analogous to a natural organism exploring a complex world which is constantly changing.

2.3.2 How Learning Classifier Systems work

The internal rules in an LCS have an 'IF…THEN' structure, and are represented in the form <condition>:<action>. Each possible action is coded as a binary string, and represents a set of values for the system's environmental effectors. A condition is represented with a ternary string over the alphabet {0, 1, #}, where # is a 'wildcard' that stands for 'don't care' (i.e. either 0 or 1). The LCS accepts messages from its environment in the form of binary strings, representing values for the array of environmental detectors, and compares these messages with the condition string of each classifier. A classifier is 'matched' if its condition string is identical to the message apart from wildcards – for example, the message 1100 could match conditions 1100, 1##0, #100, and so on.

The classifiers matched by a given message are subdivided into 'match sets' advocating different actions to take. The LCS will have some system for deciding which action should be selected; this might involve choosing the classifier with the highest fitness value, or taking the action suggested by the greatest number of classifiers. The chosen action is then sent to the environment, and if the results are desirable, the environment sends a reward to the action set (the match set containing all the classifiers which suggested that action). Classifiers which have proved useful therefore acquire higher fitness values and are more likely to be used again. A Genetic Algorithm is employed at intervals to modify the classifiers and introduce new ones in the manner previously described.

In the case of the GABIL system, the 'action' on the right hand side of the rule is simply the class that is assigned to examples which match the left-hand side. This class prediction is sent straight to the environment, which responds by telling the system whether or not it classified the example correctly – this is known is a 'stimulus-response' type system. Other classifier systems are more complex, passing messages via an internal memory before an action is finally sent to the environment. The type of actions that are possible depend on the task being performed. For example, in the animat problem, an agent must learn how to navigate around obstacles whilst in search of a goal such as food or water. The actions in this case represent which direction the agent should move in next.

2.3.3 Types of Learning Classifier System

The Michigan approach demonstrates so-called 'dynamic' behaviour which is rather more complex than the reflex behaviour seen with GABIL. A Michigan-style LCS has an internal state where multiple layers of computation may be performed before a response to the environment is generated. This is implemented by having a 'message list' where signals from the environment are stored before being matched against classifiers. If a classifier is matched, it can also place a message on the message list rather than sending an action straight to the environment. In all but the simplest systems, a mechanism is required for preventing the message list from 'overflowing'; any matching classifiers must make a 'bid' which is based on their fitness value, and the classifier making the highest bid is selected for activation. Additionally, when it is decided that an action will be sent to the environment, the reward (if appropriate) is not only given to the classifier which sent the action, it is distributed amongst all the classifiers that participated in the sequence of messages which eventually called it. In this way, successful chains of classifiers are able to acquire high fitness values.

Since the GA in a Michigan-style system operates on individual classifiers, the rules are actually competing with one another. This competition can lead to useful rules being eliminated, which makes it extremely difficult to build up the rule chains needed to produce an optimal solution. Because of this, the Pittsburgh LCS takes a different approach. These systems treat an entire population of rules as an individual with an overall fitness value, and use the GA to evolve an optimal population of cooperating rules.

Using the Pittsburgh approach immediately necessitates some consideration of how the genetic operators are to be employed. Traditionally, fixed-length strings (usually binary) are used to represent points in the search space, and crossover can be applied in a straightforward manner. However, in a Pittsburgh LCS, the GA is required to manipulate entire sets of rules which are concatenated into a long string. Since different sets might not contain the same number of rules, the two parent strings may be of varying lengths – this could be a problem if a crossover point is selected which is not present on one of the strings. Moreover, there are syntactic and semantic constraints to consider; if rules are sliced and joined together in a random fashion, it is likely to result in the production of invalid rules which lack or duplicate some of the feature settings. Clearly, this is unacceptable. Smith (1980) solved this problem by slightly modifying the crossover operator, so that a separate crossover point is chosen on each parent. These points could still occur anywhere on the string (i.e. either at a rule boundary or within a rule), but in order to achieve 'meaningful crossover', Smith insisted that the two corresponding points be in the same position relative to a rule boundary. In other words, both parents must be cut on a rule boundary, or both must be cut n bits to the right of a rule boundary (though it need not be the 'same' boundary). De Jong and Spears (1991) adopt this same approach with their system GABIL, and explain how they restricted the complexity of a rule structure in order to ensure that every rule had the same fixed length. Note that the mutation operator requires no alterations and operates at bit level as usual.

The process performed by GABIL (which stands for Genetic Algorithm Batch-Incremental concept Learner) is as follows:

- The system initially accepts a single positive or negative example of the target concept, from a pool of examples.
- The GA searches the space of variable-length strings for a rule set which achieves a 100% score for this example.
- This rule set is used to predict the classification of the second example.

- If the prediction is incorrect:
    - the GA is invoked to evolve a new rule set that scores 100% for both of the examples seen.
- Else (i.e. if the prediction is correct):
    - the example is simply stored with the previous example, and the rule set is left unchanged.
- The process continues in the same way – the system predicts the classification of each new example, and reruns the GA in batch mode if the prediction was wrong.

Note that, due to the stochastic nature of GAs, a rule set with a perfect score may not always be found within a fixed amount of time (Spears 1992). The user may therefore specify an upper bound on the number of generations, and if a 100% correct rule set has not been found within this time, the GA simply returns the best one. Although this rule set will be incorrect, it is often quite accurate.

GABIL has been tested over concepts of varying size and complexity. In order to observe how the system's predictive performance changed over time, the value plotted at each time step (i.e. after the submission of each new example) was the percentage of predictions that were correct out of the last ten made. The resulting curves were compared with those of two different systems called ID5R and C4.5, and it was found that the performance of these systems varied dramatically as the complexity of the target concept increased, while the performance of GABIL was relatively unaffected by the change. These differences were due to the in-built biases in ID5R and C4.5 – for example, ID5R favours concepts that can be represented with small decision trees. In the simpler problems where these biases were appropriate, GABIL was outperformed. However, when the concepts were more complex, requiring a greater number of rules to correctly describe them, the predictive performance of ID5R degraded significantly and GABIL was shown to be far superior. GABIL's ability to perform uniformly well on concepts of varying complexity is due to its minimal system bias – there is no in-built preference for, say, shorter rule sets. Any biases are made explicit in the GA's payoff function, which is easy to change. If shorter rule sets were indeed desired, the payoff function (rather than GABIL itself) would be altered to implement this. In these particular experiments, De Jong and Spears simply awarded fitness values equal to the square of the individual's score (the percentage of predictions correct), which provided a non-linear bias towards correctly classifying all the examples. The results they obtained show that, while the overhead of using a GA is evident on simpler problems, the strength of the system becomes clear as the complexity of the problem increases.

Whilst the Pittsburgh approach has been proven to be effective, the process is extremely slow to find solutions, as the Genetic Algorithm has to operate on extremely long strings that combine all the classifiers from the parent populations. It seems that a more compact representation of the rule sets would be useful, as the performance of the GA would be improved and the system as a whole would give faster, better results. A method of transforming a large, complex structure into a simpler one, and vice versa, is therefore required. One such system is Grammatical Evolution, discussed in the next section.

2.4 Grammatical Evolution

2.4.1 What is Grammatical Evolution?

GE was first developed at the University of Limerick by Conor Ryan, Michael O'Neill and John Collins (Ryan 1998a). It is a way of evolving expressions or complete programs in any language. The language to be generated is described using a Backus Naur Form (BNF) grammar definition, consisting of a series of production rules mapping non-terminal symbols to the terminals which appear in the language. A linear genome made up of a variable number of 'genes' – each of which is represented by an 8-bit binary number – is used to control how the grammar definition is mapped to an actual expression or program; this will be explained in the next section. A Genetic Algorithm can then be used to breed the fittest genome for creating such a program. Grammatical Evolution is language independent, and can (in theory) be used to generate functions of arbitrary complexity. It has been tested with successful results on a symbolic regression problem, a trigonometric identity problem (Ryan 1998b) and a symbolic integration problem (Ryan 1998c).

2.4.2 How Grammatical Evolution works

As mentioned above, the syntax of the language to be generated is described through a grammar in Backus Naur Form. This consists of a tuple, {N, T, P, S}, where N is the set of non-terminals, T is the set of terminals, P is the set of production rules, and S is the start symbol. The terminals are all the symbols which can appear in the language, while the non-terminals can be expanded into sequences of terminals and/or non-terminals. The production rules describe how each non-terminal can be expanded, while the start symbol is a particular non-terminal from which the mapping process begins.

Ryan et al. presented the following simple example of a BNF grammar:

```
N = { expr, op, pre-op }
T = { Sin, Cos, Tan, Log, +, -, /, *, X, (, ) }
S = <expr>
P =
   (1) <expr>   ::= <expr><op><expr> | (<expr><op><expr>) |
                    <pre-op>(<expr>) | <var>
   (2) <op>     ::= + | - | / | *
   (3) <pre-op> ::= Sin | Cos | Tan | Log
   (4) <var>    ::= X
```

In the above, the symbol `::=` denotes that the non-terminal on the left-hand side of the production rule can be mapped into that which appears on the right-hand side. The pipe symbol `|` is used to denote 'or'. The definition is recursive, meaning a production rule can call itself – for example, 'expression' appears on both the left- and right-hand sides of the first production rule.

Here it is important to note that a given non-terminal may have more than one possible expansion. In Grammatical Evolution, the linear genome is used to decide which expansion to carry out whenever there is a choice. This is done by numbering each of the various possible outcomes, then reading the next pseudo-random number from the chromosome, which first involves converting the binary gene into an integer. If, for example, the non-terminal could be replaced by any of four different expansions, one of these would be selected by taking the next available number from the chromosome mod 4. This is analogous to a process in natural biology, where each gene will express a given protein, but the physical effect of the protein is dependent on the other proteins which are generated immediately before and after it. Here, each 'gene' will give the same number no matter where it is located along the chromosome, but its effect on the final program depends on which production rule it is called upon to guide.

It is entirely possible that the system will 'run out' of genes before a complete expression is generated, i.e. before all of the non-terminals have been mapped to terminal symbols. If this happens, there are two possible approaches. One solution is to declare that genome invalid, giving it a very low fitness value so that the GA is unlikely to select it for further use. The other, more 'realistic' solution favoured by Ryan et al., is simply to wrap around once the end of the genome is reached, and reuse the same genes. Any genes which are not

required can simply be ignored, although it may be desirable to 'prune' genomes with such redundant genes, as will be discussed below.

The standard genetic operators of crossover and mutation can be applied to these linear genomes, recombining the genes and thus trying out different paths through the BNF grammar definition. The expressions or programs which are generated can then be tested with whatever inputs are appropriate for the given problem, and the performance of the program determines the fitness value of its corresponding genome. Two new operators specific to Grammatical Evolution were also introduced; these are called Duplicate and Prune. The former makes a (randomly chosen) number of copies of a gene, and places the duplicates into the position of the last gene on the chromosome; this increases the presence of a particular 'protein'. The Prune operator is applied with a given probability to any individuals which do not express all of their genes; it discards any genes that are not used in the mapping process (these are known as 'introns'). The result is that crossover is more likely to recombine useful material, rather than leaving these sequences intact and slowing the process of evolution.

2.4.3 Implementations of Grammatical Evolution

The original version of Grammatical Evolution was implemented in C++, using a Lex program with pattern-matching capabilities to interpret the BNF grammar definition supplied to the system. The current version of the source code is available for free download from the Grammatical Evolution website maintained by Michael O'Neill. This code does not include a built-in Genetic Algorithm, so it is necessary for the user to obtain one separately.

In 2002, a version of Grammatical Evolution was implemented in Java by Rebecca Griffith, an MSc student at the University of Bath (Griffith 2002). The ultimate goal was to enable GE to be applied to data mining, where previously unknown correlations are extracted from a data set. The Java version of GE, which did incorporate a Genetic Algorithm, was tested on the symbolic regression problem that was used to test the original implementation (Ryan 1998a). Although the suggested solutions generated by the system were reasonably good, the actual solution was not found. This was attributed to the limited number of tests carried out due to time constraints, and also to the lack of information known about the parameters used by Ryan et al. It was felt that the parameters chosen in this case might be leading to premature convergence, where the GA focuses quickly on non-optimal individuals that nevertheless have high fitness values.

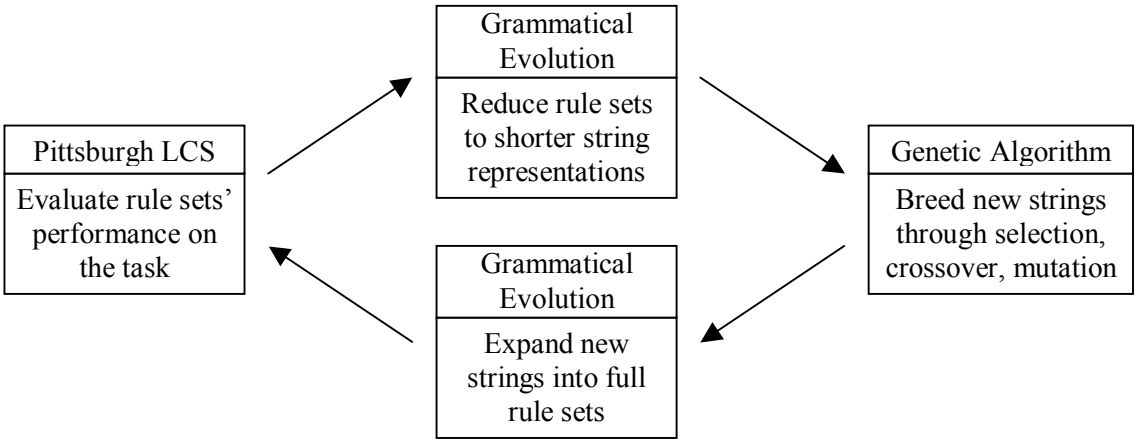
A number of potential problems were also identified with the Java implementation, relating to its lack of validation techniques (enabling illegal inputs to be received) and inefficiency in parts of the code structure. Additionally, the Java version was somewhat limited and did not attempt to implement the GE-specific Duplicate and Prune operators that were used in the original C++ version.

Although parts of Griffith's code are arguably easier to follow than their counterparts in the original implementation, it was concluded that the Java system was not robust enough to be used effectively, as changes to the BNF grammar required too many alterations to the program. If the version of Grammatical Evolution used for the project is to be based on that of Griffith, it will therefore be necessary to implement some of the suggested improvements to that system. Given the time constraints, it may be more sensible to use a version of the original implementation by Ryan et al., although it should be noted that programs written in this language can be far more difficult to debug. The problem of choosing appropriate parameters for the testing of the system will be present in either approach.

2.5 The Project

David Goldberg noted that the nature of a Genetic Algorithm's power is similar to the process of human innovation; in particular, the recombination of material via the crossover operator effectively creates new ideas by juxtaposing different ones that have worked well in the past. In that same spirit, it is hoped that this project will produce an effective new learning system by combining two fields of research that have independently enjoyed successful results.

The hypothesis, put forward by Alwyn Barry, is that Grammatical Evolution could be integrated within a Pittsburgh Learning Classifier System in order to improve its performance. As explained in section 3.3, a current disadvantage of the Pittsburgh approach is that the process of finding optimal solutions is relatively slow, due to the large size of the rule sets upon which the Genetic Algorithm operates. Since the GA would work faster upon shorter length individuals, it is suggested that the rule sets be encoded into a simpler format before being passed to the GA for the necessary processing. These shorter strings could then be expanded back into full rule sets using Grammatical Evolution. The diagram below summarises the structure of the proposed system.



There is still much research to be done into ways of improving the efficiency of Pittsburgh Learning Classifier Systems, and the use of Grammatical Evolution is just one possibility. Since it has not been tried before, it is difficult to predict how successful the project will be. It seems likely that any performance advantage gained by the new system will be more noticeable on problems of greater complexity – that is, problems where a greater number of rules and/or relevant features are needed to describe the concept, leading to manipulation of very long strings by the GA. If too simple a concept is chosen for testing the system, it may be that the compression achievable through Grammatical Evolution is limited and the system will not show any real improvement over the original LCS.

Although the literature survey has touched upon a number of extensions that are designed to increase the power of Genetic Algorithms (such as messy GAs, niching, etc), for the purposes of this project it will be important to keep the system structure as simple as possible. This is partly due to time constraints that prevent the exploration of too many different parameters, and partly to avoid complicating the system unnecessarily, which would increase the risk of errors and misleading results. Therefore, only the basic genetic operators such as roulette wheel selection, single- or two-point crossover, and simple mutation will be employed. An investigation into the best settings for population size, crossover rate and so on will be carried out during the design and/or testing phase. These parameters will likely be affected by the type of test data that is chosen. One or more appropriate data mining tasks will be identified for testing purposes; these might include the breast cancer diagnosis domain used for testing GABIL (De Jong 1993) and GENIFER (Llorà 1999), and/or the Monk's Problems.

Chapter 3

Requirements

3.1 Introduction

The literature review discussed the main reason for using the Pittsburgh approach to Learning Classifier Systems – namely that a set of cooperating rules can be evolved without the risk of useful classifiers being eliminated through competition – and pointed out that a current problem with Pittsburgh systems is their inefficiency, as the individuals used are inevitably large and so it can take a long time to find good solutions. Having established the problem and proposed ways of resolving it, the first step of the project was to create a basic Pittsburgh LCS upon which to experiment. The ultimate aim would be to develop several versions of the system, trying out different modifications; the performances of each version could then be compared, in order to determine whether the additional features or algorithmic changes led to any noticeable improvement (or otherwise) in the system's efficiency and/or effectiveness.

This chapter discusses the requirements that were gathered for the standard system. These are divided into two sets – requirements based on consideration of the project constraints, and required features of the actual system. Although the source code for an existing Pittsburgh LCS – De Jong and Spears' GABIL – was available for use, it was felt that the system developed for this project should be considerably simpler, and should also be easier to modify. The GABIL code was itself written for research purposes only, and includes many experimental versions of the same functions, as well as making use of some specialised operators ("adding alternative" and "dropping condition" – see De Jong 1993) which were considered to be an unnecessary complication for the purposes of this project. Whilst these two operators have been shown to improve GABIL's predictive accuracy in

certain situations, they would nevertheless add the complexity of extra parameters to the system, and it was reasoned that the 'standard' version of the system need not be especially advanced since the project would merely focus on whether or not the new modified versions showed any performance gains. Furthermore, the GABIL code was written in C, which was rejected as the language of choice for this project (see chapter 4 for the reasons behind this decision).

It was therefore decided that a Pittsburgh LCS should be written from scratch, as it could be constructed to be as minimal as possible, and modifications would be easy to make due to complete familiarity with the code. This decision was made with some caution as the highly iterative nature of such a system makes it sensitive to subtle algorithmic changes and therefore difficult to fine-tune; if necessary, existing GA code would be sought out and integrated within the system (though ultimately this was not needed). The requirements for the new system were based upon the understanding gained from the literature survey, and from study of the existing code for GABIL as well as XCS. The latter is a Michigan approach LCS (Wilson 1995), but its data structures and genetic operators could nevertheless provide inspiration for this system.

3.2 Project Requirements

Firstly, it was important to consider the constraints that would necessarily be placed upon the project – the obvious one being the limited time available. The standard version of the LCS would need to be developed fairly quickly in order to allow time for the experimentation and writing up of observations. With only a few months available in total and the added pressures of other undergraduate study to bear in mind, this meant that the system would need to be as basic as possible, making it straightforward to debug and minimising the risk of undetected errors. The project requirements relating to this issue can be summarised as follows:

- The system must contain no unnecessary functionality.
- A suitable programming language that is already known must be chosen.
- Each new component must be tested immediately to ensure that errors are easily traced before the system is made more complex.

3.3 System Requirements

Keeping the goal of simplicity in mind, the following functionality was identified as being essential to the system.

3.3.1 Data instances

- The system must accept a set of instances, where an 'instance' is an already classified example of the concept to be learned.
- An instance must be represented as a 'message' string which can be matched against the condition string of each classifier, and a 'class' string indicating which class the instance belongs to.
- It must be possible to supply the system with different sets of instances for different problems.
- There must be a data structure that holds all the instances to be used.
- The order of instances supplied to the system should be shuffled randomly to avoid accidentally introducing any bias within the training data which might affect the way the system works.

3.3.2 Rule sets

- There must be an object or structure representing an individual in the population. In a Pittsburgh LCS, an individual is a rule set, i.e. a set of classifiers.
- Every individual rule set must have its own associated fitness value.
- It must be possible to evaluate a rule set's performance on a classification task and assign it an appropriate fitness value based on this.
- A rule set must be re-evaluated if it undergoes crossover or mutation, since one or more classifiers may have changed.
- It should be easy to change the way in which fitness is calculated (i.e. the "payoff function"), for the purposes of experimentation.
- The classifiers within a rule set must be able to be concatenated into a single long string for manipulation by the Genetic Algorithm.
- The system must initialise the first generation rule sets with random classifiers, i.e. rules consisting of randomly generated bit strings.

- There must be a data structure that holds all the individual rule sets in the current population.
- The system must keep track of the best rule set found in each generation, and the "best ever" rule set that has been seen so far. The distinction is important since a particularly good rule set may nevertheless not survive to the next generation.

3.3.3 Classifiers

- There must be an object or structure representing a single classifier within a rule set. (Note: an alternative approach would be to work only with long strings representing the concatenated rules. The principal reason for including a separate classifier object is flexibility – it can be useful to assign values to the distinct rules themselves, as will be seen later in the project.)
- A classifier must consist of a condition string (which may contain wildcards) and an associated action string. In this system, the action string will represent a prediction as to the class of an instance that matches the condition string.
- It must be possible to initialise a classifier with random bits.
- It should be possible to specify the 'generality', i.e. the probability of a bit being initialised as (or mutated into) a wildcard.
- Wildcards must never be allowed to appear in a classifier's action string.
- It must be possible to determine whether or not a classifier is matched by a given data instance.

3.3.4 Genetic Algorithm

- There must be a way to create a new population using fitness-proportional selection over the existing population.
- It must be possible to perform crossover on a pair of parent individuals.
- It must be possible to mutate a specified bit within a rule set, replacing it with a different legal value that is chosen at random.
- The probabilities of crossover and mutation being applied to an individual rule set must be specified as system parameters.
- There must be a parameter specifying a maximum number of generations after which the Genetic Algorithm will terminate.

3.3.5 General / Other

- The system must include a suitable pseudo-random number generator. This is needed for shuffling data instances, initialising rule sets, and within the Genetic Algorithm.
- System activity must be output to a number of text files for analysis by the user.
- It should be possible to easily alter system parameters such as population size, length of message and class strings, initial number of classifiers in a rule set, number of instances to read in, crossover and mutation rates, etc.

This breakdown of required functionality helped to focus the design of the system, identifying the major data structures that would be needed and suggesting a logical separation of system components – for example, the use of a global parameters file so that constant values such as the length of a rule could be easily altered between experiments without needing to make changes to several sections of the code. Another observation is that it would make sense for the various genetic operators to be grouped together, and kept separate from unrelated functions such as those concerned with handling the data instances. Since random numbers are needed in several parts of the system, it seemed a good idea to implement the pseudo-random number generator as an individual component which would be globally accessible like the parameters.

Chapter 4

Design

4.1 Overview

This chapter discusses some of the main design decisions which were made when planning the initial version of the Pittsburgh LCS. This includes justification of the programming language chosen for the implementation of the system, and the way in which generalisation would be carried out. The various individual components of the system are outlined, and the reasons for the inclusion or exclusion of particular features are explained.

4.2 Choice of Programming Language

One of the first major decisions to make during the design stage was which programming language to use for the implementation of the Learning Classifier System. Given that there was no obvious benefit in taking the time to learn an unfamiliar language for this purpose, the two practical options were Java and C. After some consideration of the features available with each and their suitability to the tasks identified in the requirements stage, Java was the programming language chosen for the project. This was preferred for several reasons.

Firstly, its automatic memory management means that Java code is often quicker to write and easier to debug than C – an important factor given the time constraints. C code is arguably more difficult to follow, typically requiring many extra lines for memory allocation and deallocation, and using pointers rather than arrays, for example. The

requirement that the system be as simple to write and modify as possible led to the conclusion that Java would be the better option; as well as tending to be more concisely written than C, it eliminates a source of common errors such as trying to access a chunk of memory that has not been allocated to the program (which, in C, would result in the somewhat unhelpful "segmentation fault" error message). Java takes care of such issues on the programmer's behalf, and the javac compiler also provides particularly helpful error messages, specifying the location of mistakes in syntax, undeclared variables and so on. This means that the programmer is able to fix this type of minor error quickly and easily, concentrating their debugging efforts on problems with the program design itself.

Secondly, Java provides excellent support for the manipulation of strings, which would certainly come in useful for this system, since the Genetic Algorithm in a Pittsburgh LCS treats individual rule sets as long strings that may be cut and spliced in various ways. Java would enable crossover, for instance, to be done simply by concatenating certain substrings from the two parent individuals. This would be considerably more awkward to implement in C, which does not even have a string data structure as such.

Stewart Wilson's XCS (Wilson 1995) is an example of an existing Learning Classifier System that has been successfully implemented in Java, indicating that the language is indeed suitable for this purpose.

A potentially significant disadvantage of using Java is nonetheless acknowledged. Its main drawback is that as an interpreted language, it cannot be as fast as a compiled language such as C; it is reported that Java is typically around 10 times slower than C. However, many Java interpreters include 'Just-In-Time' (JIT) compilers that convert Java byte-code for repeated sections into native machine code at run-time, thus providing greater efficiency after the first execution. This can narrow the gap so that Java is only around 3 or 4 times slower than a comparable C application.

4.3 The Concept Description Language

A Learning Classifier System attempts to find a 'concept description' (i.e. a set of rules) which will enable it to correctly classify each data instance. Some consideration was needed as to how this target concept would be represented, and in particular, how the system would find a minimal concept description through generalisation.

The system accepts a set of training instances, and uses the GA to find a rule set that classifies all of these instances correctly; the best rule set found can then be used to try classifying a set of test instances. Of course, the system could easily obtain a perfect score during the training phase simply by creating a set of classifiers that were identical to the data instances supplied. The problem is that in this case, the system has not identified a pattern within the training data (it has not 'learned the concept') – one could say it has simply 'memorised' the examples, and because the solution generated will lack generalisation, it is likely to perform poorly when attempting to classify the test instances (Bacardit 2003). The system should instead be capable of identifying which features or attributes are relevant to the classification task, so that previously unseen examples can also be classified according to this pattern.

There are several ways in which generalisation can be performed; two commonly used techniques are wildcards and partial matching (see Bassett 2002). It was decided that wildcards would be used in this system as the method for indicating a feature's irrelevance. For example, suppose we are considering four attributes, each of which can be represented by a single bit, but only the second attribute is used to determine which class the instance belongs to. In this case, a suitable rule set might consist of the two classifiers #0##:0 and #1##:1. This means that if the second attribute is 0, the instance belongs to class 0; if the second attribute is 1, the instance belongs to class 1; and we do not care what values the other three attributes have. This is a simple method of generalisation which is easy to implement and which can lead to concise concept descriptions.

The GABIL system uses a slightly different approach that allows for nominal features, i.e. attributes which may have more than two possible values. In this approach, a feature with k possible values will be represented using k bits, and the disjunction of these bit settings indicates which values the feature should have. De Jong and Spears give a simple example where the legal values of a feature are the days of the week, and the pattern 0111110 represents the test for that feature being a weekday (Monday OR Tuesday OR Wednesday OR Thursday OR Friday). Generalisation is therefore implemented by setting all the bits to 1 within a feature test, indicating that the feature is irrelevant since it does not matter which of the possible values it takes. Note that since GABIL still uses binary strings, this approach can be implemented in our Learning Classifier System if desired, simply by setting the 'generality' (probability of introducing a wildcard) to zero and devoting the appropriate number of bits to each attribute.

Not all datasets use attributes with nominal values – some classification problems involve real-valued attributes, for example. A Pittsburgh LCS called GENIFER (Llorà 1999) used a modified version of the GABIL representation to allow for this type of attribute; however, for simplicity it was decided that this project would focus only on classification tasks using binary or nominal features.

Finally, while GABIL divides the data instances into 'positive' and 'negative' examples of the target concept, the system developed for this project was designed to be more flexible, allowing for the possibility of more than two classes. For this reason, actions are represented as strings (whose length is specified as a parameter) rather than just a single bit.

4.4 Inputting Data Instances

Since the system would inevitably be tested on a number of different datasets, there needed to be a simple way of switching between classification problems with minimal changes to the code. The easiest way to do this is simply to have the system read in the data instances from a text file, which is always named "instances.txt".

The only other changes necessary would be updating the global parameters specifying the number of instances, the length of a message (or condition) string and the length of a class (or action) string. These parameters provide a certain degree of validation – if not enough data instances are supplied, or if any instances have the wrong length, the system will generate an error message and terminate immediately.

Each data instance should appear on a separate line within the text file, but there is no need to include a colon between the message and class strings, since the system can use the string length parameters to separate them out automatically.

4.5 System Structure

This section gives an overview of the various components that the system was divided into – in other words, the Java classes that were to be implemented.

4.5.1 Data Instances

An Instance object would be created for each data instance that was read in from the text file. Each of these objects would have a message string and a class string, with corresponding methods for returning them.

4.5.2 Classifiers

A particular classifier object would have two strings representing its condition and action. The Classifier class would have two constructor methods; one without arguments, which would initialise the classifier with random bits, and one allowing a new classifier to be constructed from a given pair of strings. In the former case, the probability of each bit in the condition string being initialised as a wildcard would be determined using the global 'generality' parameter. The Classifier class would also include a method that accepted a 'message' string from a data instance, and returned a Boolean value indicating whether or not the classifier was matched by that message. Recall that a 'match' occurs when every bit in the classifier's condition string is either the same as the corresponding bit in the message string, or it is a wildcard (i.e. a 'don't care' symbol).

4.5.3 Rule Sets

A rule set would be an individual within the population. Each rule set object would have its own set of classifiers and a fitness value. The constructor method would accept an integer argument specifying the number of classifiers that the rule set should initially have, and it would be possible to fill the rule set with randomly initialised classifiers. It would also be possible to turn the rule set into a copy of an existing one, enabling a record to be kept of the best rule sets found. For the purposes of crossover and mutation, there would be a method to return a long string representing a concatenation of all the classifiers, and conversely, a method accepting such a string and converting it into a set of classifier objects. Fitness calculations would be done within an 'evaluate' method which would test the rule set's accuracy over a set of data instances; there would also be a method for directly setting the rule set's fitness to a given value. Finally, the RuleSet class would provide methods for returning the number of classifiers and the total length (in bits) of the string of concatenated rules, and a method for printing out the set of classifiers along with its total length and fitness value, either to the command prompt or to a specified output file.

4.5.4 Population

The Population class would only be instantiated once, and would be responsible for storing, initialising and manipulating the population of rule sets. The array of data instances (and the method for reading these in from a text file) would also be stored here, so the instances could be passed to the rule sets for their evaluation. Variables would keep track of the current generation number, the best rule set found in this generation, and the best ever rule set found so far. An instance of the genetic operators (similar in nature to a utility class, providing a collection of algorithmic services) would be held in the Population class, and rule sets would be passed to these operators as appropriate. A method called 'adapt' would contain the loop which repeatedly ran the Genetic Algorithm and then output the statistics for the new generation of rule sets, until the predefined generation limit was reached.

4.5.5 Genetic Algorithm

The GeneticOps class would contain methods for each of the genetic operators – see the next section for a list of the operators chosen for inclusion. This class would be instantiated once and its methods would act upon rule sets within the population as previously described.

4.5.6 Random Number Generator

Since a number of classes would require an instance of a pseudo-random number generator, it was decided that this should be located in its own RandomNumbers class, so that a different randomisation algorithm could be 'plugged in' if desired without needing changes to several parts of the system. A reference to the random number generator would be obtained by calling the public 'getGenerator' method in this class. The first time this method was called, a new instance of the generator would be created, but on subsequent calls it would merely return a reference to the existing instance. The choice of pseudo-random number generator is discussed in section 4.7.

4.5.7 Output Writers

As with the random number generator, the various PrintWriters were placed in their own OutputFiles class since several other classes would require access to the same output streams. The public 'getWriters' method would allow another class to obtain a reference to

the set of output files, and again, this method would only call the constructor on the first occasion. The constructor method would be responsible for initialising the PrintWriters and adding an introductory message to certain output files. Once another class had gained access to the instance of the OutputFiles class, the specific output streams required would be obtained by calling the 'getWriter' method with the PrintWriter name supplied as a string argument. The OutputFiles class would also include a method for closing all the files. A description of the various output files used by the system can be found in section 4.9.

4.5.8 Parameters

The Parameters class contains the set of constants that are used throughout the system. The public 'getParams' method would allow another class to obtain the set of parameters, creating a new instance of the class on the first call and returning a reference to this instance on subsequent calls. Each parameter would have its own corresponding method to return that value. Section 4.8 contains a full list of the parameters used.

4.5.9 Top Level Control Program

The main method would be located in the LCS class. This controller class would be responsible for instantiating the OutputFiles and Population classes, calling the population's 'adapt' method to evolve an optimal rule set, and then calling the method that closes all of the output files.

4.6 Genetic Operators

Although there is much research into how to make Genetic Algorithms more effective (e.g. Goldberg's Messy GAs as described in the Literature Review), for the purposes of this project it was believed that a "simple GA" would be sufficient. Simple GAs are commonly used by experimenters to find a baseline for the performance of a learning system; they are well understood and so any improvements to the system can be easily analysed. The genetic operators in our system would therefore be limited to selection, crossover and mutation. Since many GAs do not include Holland's inversion operator, it was decided that this one would not use it either. There would also be no extra operators such as AA and DC from GABIL (De Jong 1993).

4.6.1 Selection

There are a number of techniques for selecting individuals that will contribute to the next generation (either being copied across directly, or used as a parent for crossover). These techniques include roulette wheel selection and tournament selection. The former approach is so named because it is analogous to spinning a weighted roulette wheel, where individuals of higher fitness are given a larger slot on the wheel, increasing their chances of being selected. Tournament selection involves randomly selecting pairs of individuals and using some criterion to determine which is better (e.g. the one with a higher fitness value); this is repeated several times to create a 'selection pool' of the winning individuals, and then ordinary roulette wheel (or totally random) selection can be used to pick an overall winner from this pool. Tournament selection may therefore take longer, but it enables different selection criteria to be used apart from just the fitness values of the individuals.

It was decided that simple fitness-proportional selection with roulette wheel sampling would be employed in the initial version of the system. Melanie Mitchell explains how this technique can be implemented (Mitchell 1996):

1. Calculate the cumulative fitness (sumFitness) of individuals in the population.
2. Repeat until enough individuals have been selected:
    a. Generate a random number in the range [0, sumFitness).
    b. Loop through the individuals in the population, summing their fitnesses, until the sum is greater than the random number.
    c. Select the individual whose fitness put the sum over this limit.

The relevant method for this can be summarised as follows.

rouletteSelection: RuleSet[]→ RuleSet[]
Precondition: none
Postcondition: selects rule sets from the old population using roulette wheel sampling, and returns this mating pool.

4.6.2 Crossover

When designing the first Pittsburgh Learning Classifier System, LS-1, Smith observed that a modified version of the crossover operator was needed when using variable-length individuals (Smith 1980). There were two reasons for this. Firstly, it is insufficient to choose the same crossover point for both parents, since the individuals may be of different length and the chosen point may not exist on one parent. Secondly, the crossover points

cannot be chosen arbitrarily on both parents, as this may result in the creation of invalid classifiers with too many – or not enough – attribute values. Smith solved the first problem by choosing a separate crossover point on each parent. In order to ensure that crossover produced valid individuals, he insisted that these crossover points be in the same position relative to some rule boundary, though not necessarily the same boundary.

This 'semantic crossover' operator has since been used in other Pittsburgh systems such as GABIL, allowing the number of classifiers in a rule set to grow or shrink whilst the classifiers themselves remain at the same fixed length. GABIL selects two crossover points on each parent, making sure that each point is correctly aligned with the corresponding point on the other parent, and then swaps the segments lying between the cut points. It was decided that the LCS developed for this project would do the same. Note that crossover can occur at any point within a rule and is not restricted to rule boundaries.

The crossover method would therefore work as follows.

twoPtCrossover: RuleSet, RuleSet →
Precondition: none
Postcondition: picks two random crossover points on each 'parent' rule set, swaps the middle segments, and replaces the parents with the resulting 'child' rule sets.

4.6.3 Mutation

It was decided that standard mutation would be used, with a parameter specifying the per-bit probability of the operator's application. Note that alternative strategies exist; for example Bacardit and Garrell (Bacardit 2003) specify the probability of an individual rule set being mutated, and then select a random gene to mutate within any individual selected for this.

Another decision to be made regarding the mutation operator was whether or not its application should be guaranteed to change the value of a gene. In our system, a new value will indeed always be chosen when the mutation function is called. If the gene currently has value 0 or 1, then it may either be 'flipped' or (if it is within a condition string) mutated into a wildcard; the probability of the latter is specified by the parameter 'generality' (which is also used to determine the average number of wildcards appearing in the initial population). If a wildcard is to be mutated, it has an equal chance of becoming 0 or 1.

The mutation method is described as follows.

mutate: RuleSet, int →

Precondition: integer supplied must be a valid position within the rule set.

Postcondition: assigns a randomly chosen alternative value to the specified bit.

4.7 Random Numbers

A pseudo-random number generator, or PRNG, is required in numerous parts of the system: for shuffling the data instances, initialising or mutating bits within rule sets, selecting individuals for the next generation, applying the genetic operators with a given probability, and choosing crossover points. The number generator recommended by Bacardit and Garrell is the Mersenne Twister, which was designed to overcome flaws in various existing PRNGs and is said to be one of the best currently available. It has also been found to reduce the fluctuation of results obtained with Genetic Algorithms. The original Mersenne Twister was developed in C by Makoto Matsumoto and Takuji Nishimura (Matsumoto 1997), but implementations in a range of programming languages are now available online. A Java version written by Sean Luke was obtained from the Internet (Luke 2000); documentation is also available on the website.

4.8 System Parameters

The following were identified as global constants that would be set within the parameters file.

- *condLength* – the length (in bits) of a data instance's message and the corresponding condition string in a classifier.
- *actLength* – the length of a data instance's class string and the corresponding action string in a classifier.
- *ruleLength* – the total length of a classifier, calculated by adding together *condLength* and *actLength*.
- *numInstances* – the number of data instances to read in.
- *initialNumRules* – the number of classifiers with which to initialise each rule set.
- *generality* – the probability of a bit being initialised as, or mutated into, a wildcard.
- *popSize* – the total number of competing rule sets in the population.
- *crossoverRate* – the probability of applying crossover to a pair of rule sets.

- *mutationRate* – the probability of a bit within a rule set being mutated.
- *maxGenerations* – the maximum number of iterations that the GA can make.

4.9 Output Files

The system would generate five output files in total:

- "stats.txt" – this file contains a print-out of the entire population of rule sets and their fitnesses at each generation. It also lists the best rule set found in the current generation and the best ever rule set seen so far, as well as stating the average fitness of rule sets in this generation. The stats file is useful for analysing the quality of the solutions generated by the system.

- "log.txt" – contains a detailed log of system activity, regarding the evaluation of rule sets and the manipulation of these individuals by the GA. It is mainly used for debugging purposes and is of limited interest during analysis of experiments.

- "avgfit.csv" – each line in this text file is of the form "generationNumber, averageFitness". Arranging the data in this way allows it to be input to a graph-plotting application such as GNUplot (the tool selected for use in this project) in order to observe how the average fitness of the population changes over time.

- "bestfit.csv" – similar to the average fitness file, but this time keeping track of the best fitness in each generation.

- "worstfit.csv" – lists the fitness of the worst individual in each generation.

4.10 Fitness Evaluation

We have mentioned that the RuleSet class would include an 'evaluate' method where a fitness value would be calculated, based on how well the rule set was suited to the task of classifying the data instances. These fitness values will influence which rule sets are selected by the Genetic Algorithm, so it is important that the right kind of individuals are rewarded. However, there are many ways in which the fitness calculation could be done,

and it is not immediately obvious which way is best. When designing GABIL, De Jong and Spears chose to keep the fitness function simple by considering classification performance only, ignoring such factors as the rule set's length and complexity. Having tested an individual *i* on the set of training instances, they would use the following straightforward formula:

$$\text{fitness}_i = (\text{percent correct})^2$$

They explain that this provides a bias towards correctly classifying all the examples, whilst providing a non-linear differential reward for imperfect rule sets. In other words, the chosen fitness function encourages the rule sets to be complete (recognising all positive examples of the target concept) and consistent (recognising no negative examples). Since this project was to take a similarly minimalist approach, De Jong and Spears' fitness formula was used for the initial version of the system. However, it was later found that simply squaring an individual's predictive score did not produce satisfactory results, and chapter 6 will explain how the fitness calculation was altered to apply more appropriate pressures.

4.11 Optimisation

Early prototypes of the system proved extremely slow to run, so a number of algorithmic changes were made in order to increase the efficiency of the code. These changes are detailed below.

4.11.1 Re-evaluation

The original version of the system used a loop which would run the Genetic Algorithm to generate a new population, then evaluate all the individuals in this population and output the relevant statistics. However, many of these evaluations were likely to be unnecessary, since any rule set that was copied directly from the previous generation without undergoing crossover or mutation would be given the same fitness value as before. Therefore the code was restructured so that individuals would only be re-evaluated if they were crossed with another individual or if they had any bits mutated. This would remove a considerable amount of unnecessary processing, since evaluation involves supplying a potentially large number of data instances to a rule set and trying to match each one against every classifier.

4.11.2 Macroclassifiers

Macroclassifiers were used in Wilson's Michigan approach classifier system, XCS. He used the term "microclassifier" to refer to a rule in the ordinary sense, while a macroclassifier was the unique version of a rule, having a numerosity value to indicate how many copies of this rule were present. Macroclassifiers were initially introduced into our system for convenience when analysing the output files, since a rule set could be printed with these numerosity values instead of duplicated classifiers. However, it was later observed that if data instances were matched against the macroclassifiers instead of the ordinary classifiers, it would reduce the computation time since duplicate rules would only be considered once.

Note that a set of macroclassifiers was added to each rule set without replacing the original microclassifiers. This decision was based on consideration of how crossover is performed. Although it would be possible for the system to work with macroclassifiers alone, using the numerosity values to simulate a block of identical rules for the purposes of crossover, this would have an effect on how genetic material was distributed throughout the rule sets. Because all the copies of a given classifier would be grouped together in one section of the rule set, their separation during crossover would be less likely than if they were spread out along the length of the string, and this may result in a good classifier failing to be transferred into other rule sets. It was therefore felt that the full and unordered set of microclassifiers should be kept for use during crossover, while an equivalent set of macroclassifiers would be updated whenever necessary and used for all other processing and output.

4.11.3 Output suppression

The stats file was intended to output a record of the population at each generation. However, this is not very practical in general, since a typical classification problem will require a population of about fifty rule sets and several hundred iterations of the GA. This would obviously result in a stats file taking up a large amount of disk space, which would slow down the running of the program and be fairly difficult to read through anyway. Since it is sufficient to examine the state of the population "every so often", an extra system parameter *outputFrequency* was introduced, specifying the length of an output suppression cycle (measured in number of generations). For example, setting this parameter to 100 would mean that the population was written to the stats file once every hundred generations. Note that this output frequency does not apply to the various CSV files, since

the fitness values at every generation must be recorded in order to plot the trends accurately. Since the log file is primarily of interest for debugging purposes, all output to this file is generally suppressed during experiments by commenting out the relevant code.

Chapter 5

Test Data

5.1 Overview

This chapter briefly describes the classification problems which were selected for the testing of the system. As explained in the previous chapter, it was necessary to select datasets using only binary or nominal feature values. Two standard data mining tasks are the multiplexer problems and the Monk's problems, different versions of which can be used to test different aspects of the system's abilities. For all of our experiments the system was to run in batch mode, where a training set of examples is used to generate and refine the rule set, after which no further learning takes place (and the best solution is tried out on the test instances). This evaluation approach is in contrast to incremental mode, where learning never stops and the system constantly refines its concept description as new examples are seen. Note that datasets from real-world domains may be obtained from the online UCI Machine Learning Repository (Blake 1998); there was not sufficient time within this project to test the system on real data.

5.2 The 6-Multiplexer Problem

A multiplexer, or MUX, is a logic device used to select between different binary signals in combinatorial circuits. It has n control inputs, 2^n data inputs, and a single data output. The binary values placed on the control (or 'select') inputs are used to determine which of the data input lines is routed to the output.

The 6-multiplexer problem uses 2 control bits to specify the address of one of 4 data input lines. Using a 6-bit binary string to represent the input (that is, the 'message' part of each instance), the two leftmost bits make up the control signal, and the remaining four bits are the data inputs numbered 0 to 3 from right to left. In this problem, the 'class' of the instance is the bit value at the specified address. For example, if the control bits showed the binary address 10 and the data inputs read 0101, the output would be selected from input line 2 (that is, the third bit from the right), so the class of the instance would be 1.

The maximally general concept description for the 6-mux problem therefore consists of the following eight classifiers:

```
00###0:0      10#0##:0
00###1:1      10#1##:1
01##0#:0      110###:0
01##1#:1      111###:1
```

5.3 The 11-Multiplexer Problem

This problem works in exactly the same way as the 6-multiplexer, but this time 3 control bits are used to index into one of 8 data input lines. In this case, a total of sixteen classifiers are required to describe the target concept in its most general form.

The classifiers sought in the 11-mux problem are:

```
000#######0:0      100###0####:0
000#######1:1      100###1####:1
001######0#:0      101##0#####:0
001######1#:1      101##1#####:1
010#####0##:0      110#0######:0
010#####1##:1      110#1######:1
011####0###:0      1110#######:0
011####1###:1      1111#######:1
```

5.4 The Monk's 1 Problem

The Monk's problems (Thrun 1991) are binary classification problems set within an artificial robot domain, where robots are described by six different attributes. The attributes and their possible values are as follows.

a1 (*head shape*):

1 – round

2 – square

3 – octagon

a2 (*body shape*):

1 – round

2 – square

3 – octagon

a3 (*is smiling*):

1 – yes

2 – no

a4 (*holding*):

1 – sword

2 – balloon

3 – flag

a5 (*jacket colour*):

1 – red

2 – yellow

3 – green

4 – blue

a6 (*has tie*):

1 – yes

2 – no

There are three Monk's problems designed to test different capabilities of a concept learning system. The simplest of these problems is Monk's 1, where the system must learn a straightforward Boolean expression which can be optimally represented by a set of twelve classifiers. This problem tests the system's ability to generalise, since only three of the six attributes are relevant for the concept description.

The target concept for Monk's 1 is (a1 = a2) OR (a5 = 1). In other words, a given robot is a positive example of the concept if its head shape and body shape are the same, or it has a

red jacket. Saxon and Barry (Saxon 1999) describe a suitable encoding strategy that was used for testing XCS on this problem; each attribute is represented by a minimal corresponding binary string, e.g. the possible values 1, 2 and 3 would be encoded as 00, 01 and 10. This means that a ten-bit string is required to represent all six attributes, plus a single bit for the predicted class of an instance (1 for a positive example, 0 for a negative example).

The optimal set of twelve classifiers for this problem using the binary encoding scheme is as follows (note that the order of the attributes reads from right to left):

```
#######1#1:1      ##1###1#0#:0
######0000:1      #00#######:1
######1#1#:1      #1#####0#1:0
##1####0#1:0      #1#####1#0:0
##1####1#0:0      #1####0#1#:0
##1###0#1#:0      #1####1#0#:0
```

5.5 The Monk's 2 Problem

The second Monk's problem is more challenging; this time the concept description requires a complex set of disjunctions of conjunctions. The system must therefore be capable of maintaining a set of many cooperating classifiers in order to represent the classification accurately. In this case the target concept is that exactly two of the attributes have the first of their possible values. For example, if both the head shape and the body shape have the value 'round', then none of the other attributes may have their first value – so the robot is not smiling, is not holding a sword, does not have a red jacket, and does not have a tie.

For this problem, Saxon and Barry found that their original encoding scheme did not allow XCS to explore the solution space sufficiently. They were able to obtain better results by using an enumeration encoding not unlike that of GABIL. An attribute with n possible values is mapped to a binary string of length n, and the integer ranking i is represented by setting the ith lowest bit to 1 and all the others to 0. A string of length 17 is therefore required to represent the six attributes in the Monk's problems.

The optimal concept description using this encoding scheme would contain 42 macroclassifiers, since there are 15 combinations in which exactly two of the attributes have their first value (i.e. rules to classify positive examples), and 27 combinations where

more or less than two first values appear (i.e. rules to classify negative examples). All of the classifiers would have wildcards in the positions corresponding to non-first values:

```
#0###0##0#0##0###:0
#1###1##0#0##0##0:1
#1###1##1########:0
```

(and so on). However, this exact set of macroclassifiers may be extremely difficult to find, since an attribute-value pair can be represented in a number of different ways using the enumeration encoding.

5.6 The Monk's 3 Problem

The third Monk's problem tests how robust the system is when faced with noisy data. Real data will typically contain a certain amount of noise, which should not hinder the system's ability to learn the concept accurately. Therefore, 5% noise is added to the training data for this problem.

The target concept for Monk's 3 is (a5 = 3 AND a4 = 1) OR (a5 ≠ 4 AND a2 ≠ 3). The interpretation is that either the robot has a green jacket and is holding a sword, or the jacket is not blue and the body shape is not octagonal.

Saxon and Barry's experiments with XCS showed no significant difference in performance between the binary and enumeration encoding schemes for this problem. They point out that in theory the enumeration encoding should be used for all of the Monk's problems, since all attributes are categorically valued and the binary encoding is somewhat inconvenient for representing relations between attributes and logical negations. However, the binary encoding is more effective for simple concepts such as that of Monk's 1 due to being more compact.

The optimal concept description for Monk's 3 using the binary encoding scheme is believed to consist of the following ten macroclassifiers:

```
####1#1###:0      #0#####1##:1
###1##1###:0      #0####0###:1
##0####1##:1      #0####1###:0
##0###0###:1      #1000#####:1
##1###1###:0      #11#######:0
```

Chapter 6

The Bloat Effect

6.1 Overview

This chapter discusses the problems that were encountered with the first version of the Pittsburgh LCS, whose design was outlined in chapter 4. These problems were related to the uncontrolled growth of rule sets (known as "bloat") and a lack of generalisation pressure. It was hypothesised that the problems could be resolved by evaluating individual classifiers and removing any that were inaccurate or unnecessarily specific. These techniques had been successfully employed within XCS (Wilson 1995), but had not previously been tried in a Pittsburgh approach system. Since this would be an interesting area of investigation in itself, it was decided that the project would focus on this new hypothesis rather than the possibility of integrating Grammatical Evolution within the system, which would be left as a suggestion for future work. The algorithm that was developed to implement the proposed changes is described in this chapter, while the results of experimentation can be found in chapter 7.

6.2 The Bloat Problem

As explained in chapter 4, it was initially decided that the fitness of rule sets in the Learning Classifier System would be calculated by squaring the individual's predictive score obtained over the data instances. Each rule set would be presented with one data instance at a time, and a match set containing all the classifiers matched by that instance would be constructed. If no classifiers were matched by a given instance, the rule set would

simply make a null prediction and its score would not be incremented. Some form of conflict resolution was required in cases where the matched classifiers advocated different actions (i.e. there was disagreement over which class the instance should belong to). This situation was resolved by taking a majority vote; the class suggested by the most classifiers would be chosen as the rule set's overall prediction for that data instance. If no single action proved more popular than all the others, then the first of the most popular suggestions would always be chosen. An alternative strategy would be to simply pick one of the classes at random, but this was found to be an unsatisfactory solution since identical rule sets could achieve different scores over the same set of data instances. The individual's score was based on the total number of correct predictions – there was no explicit penalty for classifying a data instance incorrectly.

The system was tested on the reasonably simple 6-multiplexer problem (described in chapter 5), and it was immediately apparent that there was an unsuitable balance of evolutionary pressures. Rather than the population converging on an appropriately sized set of classifiers, the best ever rule set was seen to grow rapidly throughout the generations, typically including dozens or even hundreds of classifiers – with a wide variety of accuracies – by the time the GA terminated. This caused the system to run very slowly, and it was virtually impossible to discern any useful pattern in the solutions generated.

The phenomenon observed here is a common problem in learning systems where variable-length individuals are used (as in the Pittsburgh LCS). The tendency for the length of solutions (e.g. the number of classifiers within a candidate rule set) to grow out of control at an exponential rate is known as the *bloat effect*. It occurs when the fitness function used by the system takes into account only the 'goodness' (e.g. predictive accuracy) of the candidate solutions, and pays no attention to the size of these individuals. In such a situation, a rule set's fitness is likely to increase if it maximises its chances of correctly classifying the training instances; this results in a tendency for individuals to simply accumulate more and more classifiers. Since there will probably be more long individuals than short ones with a given fitness value, any new solutions that are constructed are also likely to be long. The final rule sets produced will frequently be much larger than necessary, and may not even be particularly good solutions; it could be that the individual has 'memorised' the training instances rather than learning the pattern, in which case the lack of generalisation will lead to poor performance in classifying the test instances (Bacardit 2003).

When designing the original Pittsburgh classifier system, LS-1, Smith carried out some mathematical analysis of his single-point crossover operator for variable-length individuals (Smith 1980). He saw that the lengths of the pair of offspring produced can range from (2 and L+L' - 2) to ($\lfloor$(L+L') / 2$\rfloor$ and L+L' - $\lfloor$(L+L') / 2$\rfloor$), where L and L' are the lengths of the parent structures. Having determined the probabilities of each length pair occurring, he found that crossover "exerts a kind of dual pressure on the lengths of structures in the population, away from both structures of highly dissimilar lengths and structures of highly similar lengths". If the two parent individuals differ greatly in length, their two offspring are likely to be more similar in length than their parents; if the parents are of similar length then crossover is more likely to produce very different length offspring. In other words, it is not the use of variable-length individuals alone which causes bloat.

Smith points out that "the selection of parent structures on the basis of performance, however, does introduce a bias with respect to structure length". Individuals with too few rules are likely to perform poorly due to having insufficient knowledge about the task domain, i.e. the complexity of the task imposes a certain "minimum threshold" on the appropriate length of individuals. If one offspring produced through crossover has a length below this minimum threshold, it is unlikely to be selected for future generations; by contrast, there is no such selection pressure against the other, longer child. Thus one would expect the average length of structures in the population to increase over time.

The experiments of Langdon on an artificial ant problem confirm that fitness-based selection is responsible for the occurrence of bloat (Langdon 1997a). He explains that in general with variable-length discrete representations, there are multiple ways of representing a given behaviour, and there tend to be many more long representations than short ones. Practical search techniques use previously discovered high fitness representations as points from which to continue the search, i.e. there is a bias in favour of representations "that do at least as well as their initiating point(s)". Since finding improved solutions becomes increasingly more difficult as the search progresses, the selection bias favours representations which have the *same* fitness as those from which they were created, and the GA may begin to randomly search for new representations of the best solution found so far. Because there are many more long representations than short ones for the same solution, such a random search will inevitably lead to bloat. This is backed by Price's Covariance and Selection Theorem.

Another possible cause of bloat is that good solutions are more likely to be preserved in a longer individual. Referring to Genetic Programming, Banzhaf and Langdon (Banzhaf

2002) explain: “Neutral code promotes survivability of a genome because it provides locations in the genome where operators that would otherwise destroy the present fitness value can hit without a damaging effect”. This can be applied to the Pittsburgh LCS; longer rule sets will contain more copies of good classifiers, increasing the likelihood of those classifiers surviving crossover and mutation.

An obvious first step in tackling the bloat problem within our system was to try penalising the individuals for incorrect predictions. Rather than base the score on the percentage of correct answers (that is, correct predictions / number of instances), the number of correct predictions would be divided by the total number of matched classifiers, so a rule set containing many inaccurate classifiers would achieve a lower fitness than one without. However, this resulted in the opposite effect to bloat – the population would rapidly converge on individuals containing only a single correct classifier, since this rule set would be 100% accurate despite not matching many data instances. Having rejected this approach, alternative ideas were investigated.

Bloat has been studied extensively within the field of Genetic Programming, where the individuals to be evolved are parse trees. Not only does the bloat problem result in larger solutions than necessary, it also consumes memory and increasingly slows down the rate at which new individuals can be evaluated and manipulated. Additionally it can hamper effective breeding if the growth consists largely of non-functional code, since this code can play a protective role against crossover (De Jong 2001). It should be noted, however, that Langdon and Poli (Langdon 1997b) suggest there are circumstances in which bloat may be beneficial; ‘introns’ (i.e. non-functional code) may be important as “hiding places”, protecting genetic material that, while not used in the current individual, is a helpful building block later on. In cases where fitness criteria are dynamic, “a change in circumstance may make it advantageous to execute genetic material which had previously been hidden in an intron”.

A number of techniques have been suggested to counter bloat. The two most popular techniques within the Genetic Programming literature are *maximal depth restriction* and *parsimony pressure*, described below. Other methods include using a fitness-based “plagiarism penalty” on programs which achieve the same score as their parents, i.e. those which fail to innovate (Langdon 1998); this was found to slow the bloat effect but did not prevent it. Langdon and Poli (Langdon 1997b) also list suggestions for tailoring the genetic operators, such as adjusting their frequency of application to make a decrease in

complexity slightly more probable than an increase, or making the likelihood of potentially disruptive genetic operations proportional to the size of the individuals.

In the most popular method of bloat control, a child created through crossover or mutation is rejected if it exceeds some maximal depth limit, and the original parent is copied to the new generation instead. A similar but less common restriction approach is *pseudo-hillclimbing*, where a new child's fitness is immediately assessed and the child is rejected if the fitness is not superior to that of its parent. Whilst this approach does not actually compare size at all, it is reasonably successful at limiting tree growth because large numbers of parents are injected directly into later generations, and parents are generally smaller than their children. The cost of this stunting effect is a loss of diversity (Luke 2002). Unfortunately the fixed size limit approach does not always scale up well, and a fundamental problem is that the maximum size must be chosen in advance, when it is often unknown. It would be preferable to have the algorithm search for individuals that are as large as necessary but not much larger.

Parsimony pressure directly includes the size of an individual as a factor in its probability of being selected. Fitness is computed as a (usually linear) function of an individual's raw fitness and its size; the main problem is that the appropriate weighting is extremely difficult to find, especially since the relative significance of the parameters may well change during the course of an evolutionary run. This can be solved by adapting the size parameters as the run progresses, but Luke and Panait point out that such techniques must usually rely on problem-specific analysis. Another issue is that if the tradeoff surface between the two fitness components is concave – that is, if better performance in one objective means worse performance in the other – then no linear weighting exists that will find the "in-between" individuals that perform reasonably well in both respects; any linear weighting will favour individuals that do well in one of the objectives only. Smith claimed to have successfully reached "a suitable compromise in the influence of various domain independent and task specific measures in the derivation of the composite performance measure" for his LS-1 system (Smith 1980), but the approach has yielded mixed results in the literature. As an alternative to seeking a single balance for parsimony pressure, "multi-objective optimisation" methods have also been used, where one explicit objective is to find compact or fast programs. However, this can result in premature convergence to small individuals unless careful measures are taken to promote diversity within the population.

Returning to use of our original fitness function, we tried introducing a preference for shorter rule sets within the Genetic Algorithm, by using tournament selection instead of

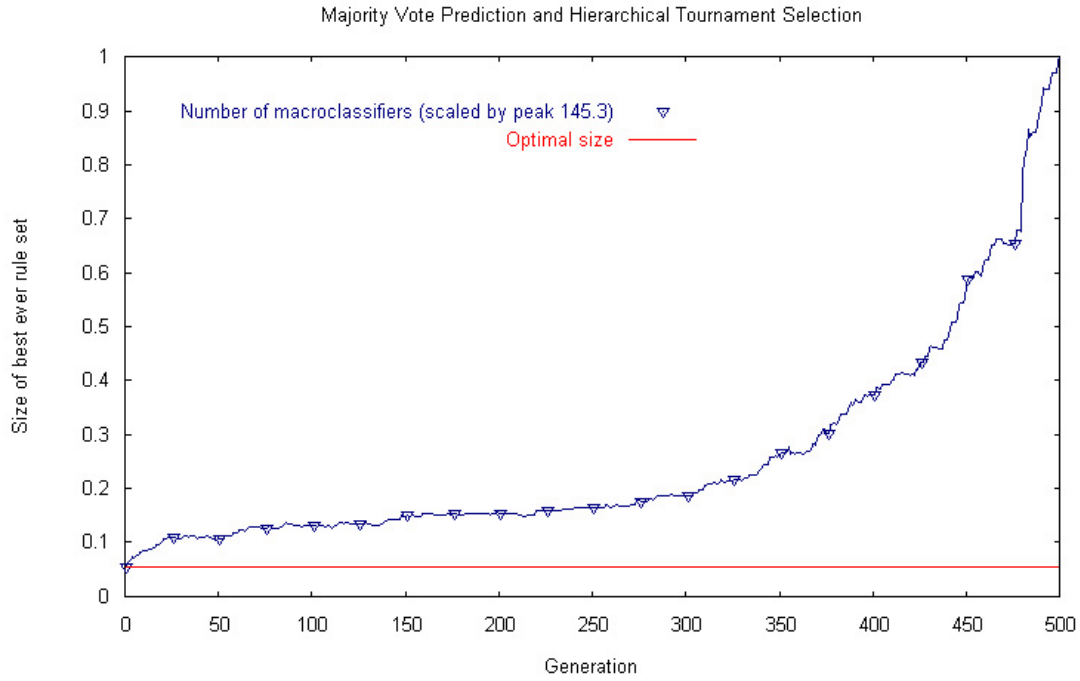
basic roulette wheel selection. Bacardit and Garrell (Bacardit 2003) describe a way to implement 'hierarchical tournament selection' which seemed appropriate for use in this system. It works as follows:

- Randomly select individuals *a* and *b*.
- If $|\text{fitness}_a - \text{fitness}_b| < \text{threshold}$
    - if one individual is shorter than the other
        - add the shorter individual to the selection pool.
    - else
        - add the individual with higher fitness to the selection pool.
- Else
    - add the individual with higher fitness to the selection pool.

This is repeated until the selection pool is large enough, and then roulette wheel selection is used to pick the individual that will be placed into the new population. The decision to switch to hierarchical tournament selection meant the introduction of two new system parameters, *fitnessThreshold* and *tournamentSize*. Additionally, a sixth output file called "size.csv" was generated so that the growth of the best ever rule set could be plotted across the generations.

Although informal experiments were carried out using this operator with a variety of fitness threshold values, there was no significant improvement in the system's performance – the longer, higher fitness individuals continued to dominate the population, despite the attempted bias toward concise rule sets.

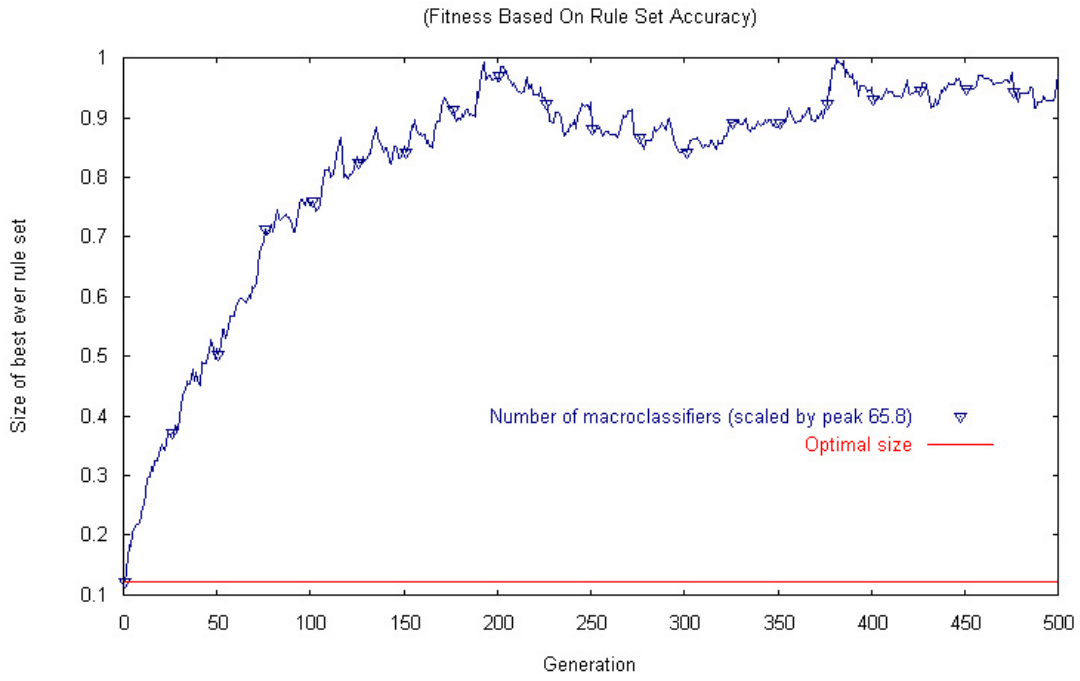
The graph below shows the growth curve for the best ever rule set, averaged over 30 test runs, when a fitness threshold of 100 was used (note that the optimal fitness for the 6-multiplexer problem is $64^2 = 4096$). As with all the experiments carried out during this project, the crossover rate was set to 0.6 and the mutation rate was 0.001. Since it is known that the 6-multiplexer target concept can be optimally described using eight classifiers, it seemed reasonable to initialise each individual with eight rules. The population size (number of individuals) here was 10. The graph shows that although the size of the best ever rule set remains controlled in the early stages, the hierarchical tournament selection proves to be an ineffective bloat control mechanism once there is a greater difference between individuals' fitnesses, and the best ever rule set begins to grow out of control.



It was found that increasing the fitness threshold parameter introduced too much bias towards shorter individuals, and the resulting solutions were small but of poor fitness. These results suggested that hierarchical tournament selection alone was not sufficient to combat the bloat problem. Nevertheless, the operator was used throughout the rest of our experiments, since it is more powerful and flexible than roulette wheel selection.

In order to strike a compromise between encouraging numerous matches and ensuring the accuracy of these matches, the fitness calculation was adjusted in the following way. Rather than using a majority vote to determine the class prediction made for a given data instance, the rule set's accuracy with regard to that instance was computed as (number of matched classifiers making the correct prediction) / (total number of matched classifiers). These accuracy values were summed, and the cumulative figure was then divided by the total number of training instances. This provides a method of penalising a rule set for containing classifiers that advocate the wrong actions, whilst still encouraging the rule set to be capable of classifying as many data instances as possible.

Although this modified fitness function appeared to slow the growth of rule sets to some degree, it was still insufficient. The growth curve of the best ever rule set was again averaged over 30 test runs, and the resulting graph shows that there is no real pressure towards finding a minimal set of classifiers to describe the concept:



Each individual was initialised with eight rules, and after 500 generations, the fittest rule set had on average grown to around eight times this size (in terms of macroclassifiers) and showed no sign of stopping. If the correct pressures were in place, one would expect to see the size of the best solution grow initially, reach a peak and then gradually shrink before stabilising at the appropriate number of macroclassifiers. Instead, here we observe that the rule set grows rapidly for the first 100 generations – since any increase in size is likely to significantly improve the fitness – and then continues to grow at a less dramatic rate as the fitness gains decrease. Overlaying a straight line on this part of the curve would confirm that the best ever rule set continues to gradually expand instead of shrinking to a more appropriate size.

It seemed that a more effective strategy would be to identify weak classifiers within the rule sets and explicitly prune them, thus improving the quality of solutions as well as providing bloat control. This would involve assigning an accuracy value to each classifier, which would not only be used to calculate the overall fitness of the rule set, but would help to guide the search for solutions more effectively. The next section describes the algorithm that was developed to do this.

6.3 The New Project Hypothesis

Rather than treating the rule sets as programs and imposing arbitrary limits on their size (as is often done in the field of Genetic Programming), it was believed that the bloat problem could be controlled by making use of the information within the Pittsburgh LCS. The new approach was inspired by techniques used within Wilson's classifier system XCS, although they were not used to tackle bloat in that system; the Michigan style LCS has a fixed population size and simply expands the numerosity of the most accurate macroclassifiers, so the bloat effect does not occur. Nevertheless, it was hypothesised that ideas from XCS could be included in a Pittsburgh style system as a way of controlling the size of rule sets, by identifying and removing unwanted classifiers.

Firstly, a new payoff function was devised to measure fitness within the various niches, where a *niche* is "a set of environmental states each of which is matched by approximately the same set of classifiers" (Wilson 1995). This is described in section 6.3.2. The classifiers themselves are evaluated, enabling relatively weak rules to be filtered out. Finally, accurate classifiers are compared with others within the same match sets, to determine appropriate generalisations and remove overly specific rules.

The process can be summarised as follows:

For each rule set
    Set raw fitness to zero
    Evaluate accuracy of each classifier
    For each data instance
        Create an action set of correct matching classifiers
        Calculate mean accuracy of classifiers in the action set; add this to raw fitness
        If action set contains a totally accurate classifier
            Delete other classifiers in the action set that are accurate but subsumed by it
    Set fitness to (raw fitness / total number of data instances)
    Delete any totally inaccurate classifiers
    Delete weak classifiers that appear in large action sets

The next few subsections provide more detailed pseudocode for the algorithm and explain the purpose of each part. The original algorithm was written by Alwyn Barry; the final version as described below includes a slight modification to fix a bug that was identified within the subsumption process.

6.3.1 Pseudocode for the new algorithm

```
For each rule set

    ruleSet.meanActionSetSize = 0
    ruleSet.rawFitness = 0

    for each classifier
        classifier.meanActionSetSize = 0
        classifier.meanActionSetAccuracy = 0

    for each classifier
        for each data instance
            if (classifier is matched)
                classifier.matches++
                if (classifier.action == instance.class)
                    classifier.correct++
        classifier.accuracy = classifier.correct / classifier.matches
        classifier.numActionSets = classifier.correct

    for each data instance

        empty the action set
        for each classifier
            if (classifier is matched AND classifier.action == instance.class)
                add classifier to action set

        actionSetAccuracy = 0
        for each classifier in action set
            actionSetAccuracy += classifier.accuracy
        actionSetAccuracy /= number of classifiers in action set

        for each classifier in action set
            classifier.meanActionSetSize += number of classifiers in action set
            classifier.meanActionSetAccuracy += actionSetAccuracy

        ruleSet.meanActionSetSize += number of classifiers in action set
        ruleSet.rawFitness += actionSetAccuracy

        carry out subsumption deletion

    ruleSet.rawFitness /= number of data instances
    ruleSet.meanActionSetSize /= number of data instances

    carry out inaccuracy deletion
```

6.3.2 Evaluation of classifiers

In the new algorithm, evaluation is first carried out at classifier level. Each classifier is assigned an accuracy value, indicating how often it correctly classifies the data instances it is matched by. A classifier may therefore be considered 100% accurate whether it matches several instances or just one. The *numActionSets* variable, however, records the total number of correct predictions that the classifier makes (i.e. the number of action sets it will appear in).

We define an action set as the set of matched classifiers that correctly classify a given data instance. Hence, there will be a corresponding action set for each instance in the training set (though some of these action sets may be empty). Since every classifier now has its own accuracy value, it is possible to calculate the average accuracy of classifiers in each action set. These mean accuracies are summed and finally divided by the total number of data instances, giving an overall fitness value for the rule set. In other words, fitness is based on the rule set's accuracy within the various niches – so rule sets with a higher number of accurate classifiers in each action set will ultimately be favoured for reproduction. This approach is better than basing fitness on a raw measure of correct classification, since there is now a penalty for classifiers that predict the same action in cases where that action is inappropriate.

Note that the average size of the action sets is also calculated for later use. (The 'size' of an action set takes into account the numerosity of the macroclassifiers, not just the number of different rules, to ensure that the rule set's resources are appropriately distributed among the possible action sets.) Additionally, each classifier keeps a record of the average size and accuracy of the action sets in which it appears. This provides a method of identifying classifiers which are not up to standard within their niches.

6.3.3 Subsumption deletion

We say that a classifier A *subsumes* another classifier B if for each bit along their length, either both classifiers have the same value or A has a wildcard. That is, A is a more general version of B. Since the Learning Classifier System is required to find the 'pattern' in a set of data, rather than simply memorising the training examples – i.e. the aim is to find the *maximally general* accurate solution – it makes sense to remove classifiers that are unnecessarily specific once the pattern is discovered. For example, if a rule set contains the classifier 00###0:0 and this classifier is 100% accurate, then a more specific classifier such

as 00##10:0 can be dropped since we already know that the fifth bit is not relevant for the classification of matching instances.

Subsumption deletion is carried out as follows:

```
highestGenerality = 0
index = -1
for each classifier in action set
    if (classifier.accuracy == 1
    && classifier.numActionSets > 1
    && classifier.generality > highestGenerality)
        highestGenerality = classifier.generality
        index = classifier index
if (index > -1)
    for each classifier in action set
        if (classifier.accuracy == 1
        && classifier subsumed by classifier[index]
        && randomNumber < subsumptionProbability)
            delete classifier
```

Here we search the current action set for the most general totally accurate classifier, if one is present, and then remove all classifiers which are accurate but more specific than this one. Note that a classifier is only eligible to subsume others if it appears in more than one action set. This is a precaution to check that the generalisation is genuinely useful, since in some data sets, not all of the attributes in a condition will be fully tested.

6.3.4 Inaccuracy deletion

The following pseudocode describes the process of inaccuracy deletion:

```
for each classifier
    if (classifier.numActionSets == 0)
        delete classifier
    else
        classifier.meanActionSetSize /= classifier.numActionSets
        classifier.meanActionSetAccuracy /= classifier.numActionSets
        if (classifier.accuracy < classifier.meanActionSetAccuracy)
            if (randomNumber < deletionProbability
            && classifier.meanActionSetSize >= ruleSet.meanActionSetSize)
                delete classifier
```

First, any classifiers that do not appear in any action sets are deleted, since this means they are completely inaccurate (i.e. they always suggest the wrong classification). In addition to this, classifiers may be removed if they tend to appear in the largest action sets and their accuracy is below average compared to other classifiers in these action sets. The effect is that action sets are reduced to approximately equal size by driving out classifiers that are inadequate. Note that two new system parameters, *subsumptionProbability* and *deletionProbability* have been introduced within this algorithm.

Chapter 7

System Testing

7.1 Overview

The algorithm described in the previous chapter was implemented and tested on the 6-mux problem. The results are presented in the next section. Initial results were encouraging, and confirmed that the techniques borrowed from XCS were effective at bloat control within a Pittsburgh LCS. However, the concept descriptions generated were sometimes slightly larger than necessary, due to the presence of 'cheat rules' that were accurate but not part of the target concept. It was found that the quality of solutions could be improved by adjusting the subsumption process; instead of simply deleting overly specific rules, they could be replaced by copies of the classifier that subsumed them. This meant that the best classifiers were able to dominate the rule set, making them more likely to drive out the cheat rules. Statistical analysis of the results obtained from the two system versions showed that the 'subsumption replacement' approach led to a significant improvement, both in the quality of the final rule sets and the speed with which an accurate solution was found. The system will be referred to as 'A-PLUS', standing for *Accuracy-based Pittsburgh Learner Using Subsumption*.

7.2 Initial Results

The system was first tested using the algorithm described in chapter 6. It was hoped that the explicit pruning of unwanted classifiers would cause the best ever rule set to decrease considerably in size, eventually leaving only the eight macroclassifiers needed to describe the 6-multiplexer target concept (see chapter 5).

7.2.1 Experimental method

A total of 30 test runs were carried out using the 6-multiplexer data instances. For each generation, the system outputs the fitnesses of the best and worst individuals, the average fitness of the population, and the number of macroclassifiers in the best ever rule set. This data was averaged over the 30 runs and plotted on graphs using GNUplot. Additionally, the final solutions generated on each run were examined to determine whether or not the optimal concept description was found. The average number of generations needed to find a rule set with perfect fitness 1.0 was also calculated.

7.2.2 Parameters

The following system parameters were used on every test run.

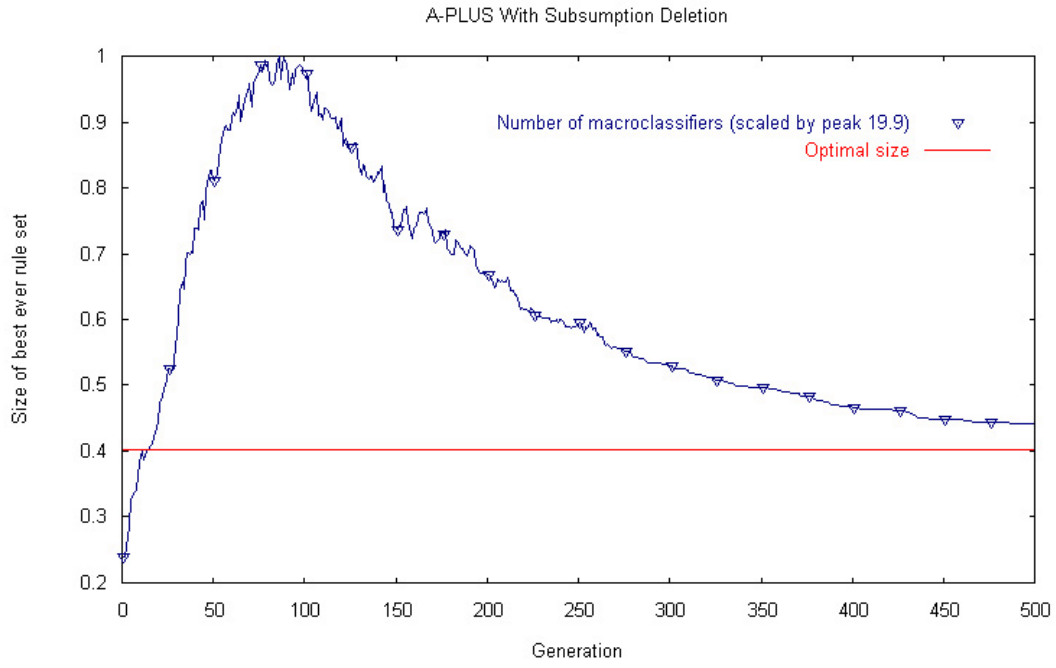
Data instance parameters:
condLength: 6
actLength: 1
numInstances: 64

Rule set parameters:
initialNumRules: 8
generality: 0.33
deletionProbability: 1.0
subsumptionProbability: 1.0

GA parameters:
popSize: 50
crossoverRate: 0.6
mutationRate: 0.001
maxGenerations: 500
tournamentSize: 5
fitnessThreshold: 0.005

7.2.3 Results

The curve obtained for the growth of the best ever rule set shows a clear improvement over previous efforts. The bloat effect has been eliminated; indeed, the average peak size of the best solution is only around 20 macroclassifiers, little more than twice the number of rules with which the individuals are initialised. After around 100 generations, the best ever rule set steadily shrinks before almost reaching the optimal number of macroclassifiers (indicated by the flat line on the graph). In fact, 18 out of 30 runs succeeded in finding the ideal concept description with eight macroclassifiers. We would not necessarily expect the most concise solution to be found every time due to the stochastic nature of the search.



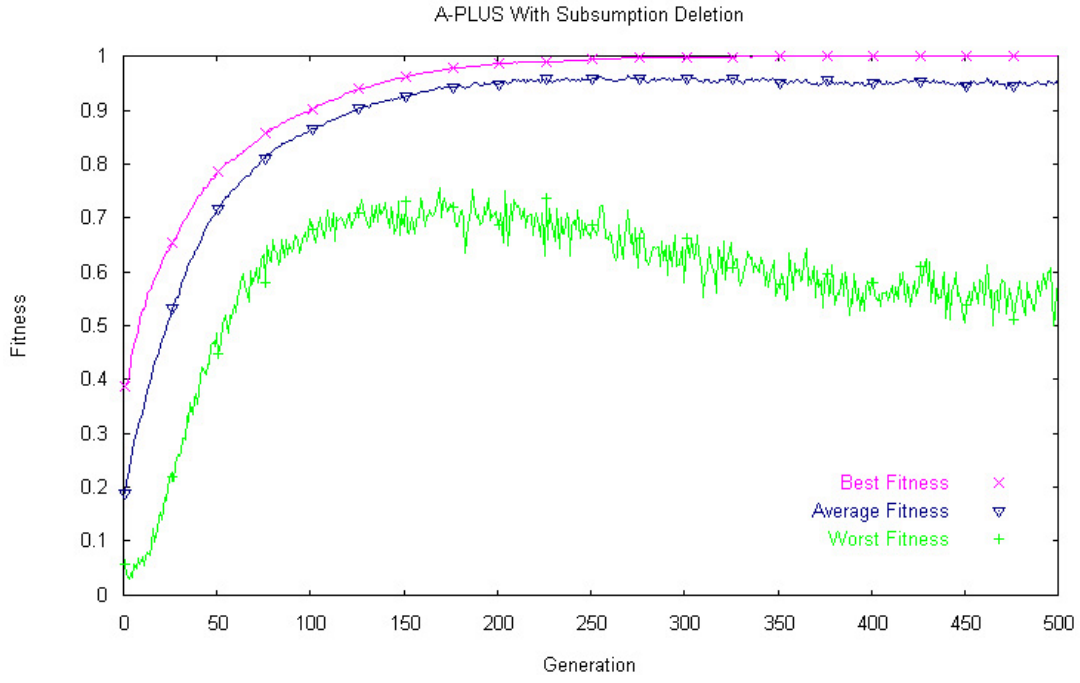
The other 12 runs all produced solutions containing between one and four 'cheat rules'. The following are examples of what is meant by a cheat rule:

```
#0#0#0:0    #10#0#:0    0###00:0    1#00##:0
#0#1#1:1    #11#1#:1    0###11:1    1#11##:1
```

Such classifiers would have an accuracy value of 1.0 since they identify a correct pattern within the data; they are, nevertheless, superfluous to the target concept description. They cannot be removed by subsumption, as they have wildcards in positions where the 'correct' rules do not. 11 of the test runs also produced solutions containing some non-maximally general rules (between one and three of them). Although these solutions were much better

than any seen with previous versions of the LCS, there was evidently still room for improvement, as the appropriate generalisations appeared not to have quite enough influence within the rule sets.

The next graph shows how the best, worst and average fitnesses changed over the generations. It took the system 203.8 generations on average to find a rule set with optimal fitness; the shortest time was 104 generations and the longest was 380 generations. The curve shows that the average fitness of the population increases fairly rapidly, and remains stable upon reaching its peak value. The deterioration in fitness of the worst individual can be attributed to the removal of a large number of inadequate classifiers, leaving very few (perhaps only a single classifier) in the rule set.



7.2.4 Conclusion

It can clearly be seen from the solution growth curve that the new algorithm had the desired effect of controlling bloat; in fact, the A-PLUS system was able to produce the ideal solution in more than half of the test runs, and came close on all of the other occasions. However, there was a tendency for accurate 'cheat rules' to survive until the end of the run, as the system had no way of distinguishing them from the desired rules. The system also had slight difficulty in finding all eight optimal generalisations, sometimes failing to recognise when an attribute was irrelevant within a rule.

7.3 Subsumption Replacement

It was thought that cheat rules and overly specific classifiers could be driven out of the best ever rule set by increasing the presence of the desired classifiers. This could be done during the subsumption phase. In the original algorithm, accurate classifiers were deleted if there was another accurate macroclassifier that was more general (i.e. had more wildcards); this was to encourage the system to find the optimal generalisations rather than memorising specific examples. However, instead of simply deleting the more specific classifiers, an alternative approach would be to replace them with copies of the subsuming classifier. This would result in more copies of the 'good' rules being distributed throughout the rule set, increasing the likelihood of these rules being preserved during crossover. Their numerosity should therefore dominate the unwanted classifiers, which would tend to be driven out over time by the GA.

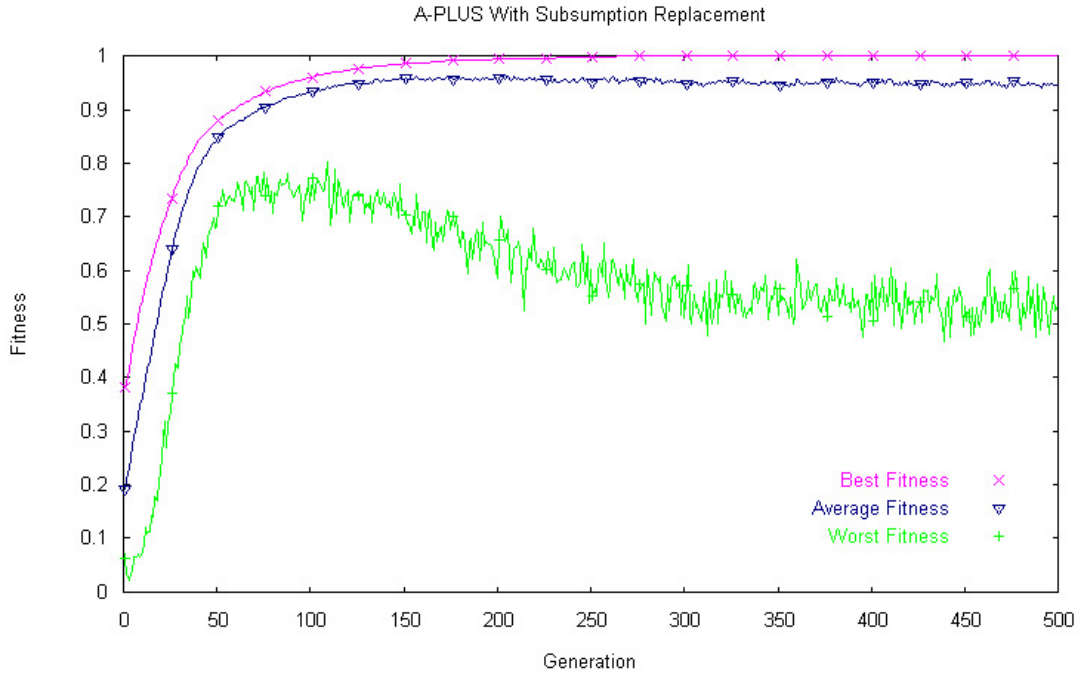
7.3.1 Experimental method

The experiments were repeated using subsumption replacement instead of deletion. The parameters of the experiment were unchanged. In addition to the previous methods of analysis, the best and average fitness data from every hundredth iteration of each of the 30 runs was entered into a spreadsheet, and a paired t-test was run on these results. The purpose of this test is to discover whether there is a significant difference between the two sets of runs due to the changes made in the algorithm. It was decided that there was too much variance in the worst fitness data to make a paired t-test worthwhile. The *null hypothesis* was that using subsumption replacement instead of subsumption deletion has no effect on the best and average fitness values within the population.

7.3.2 Results

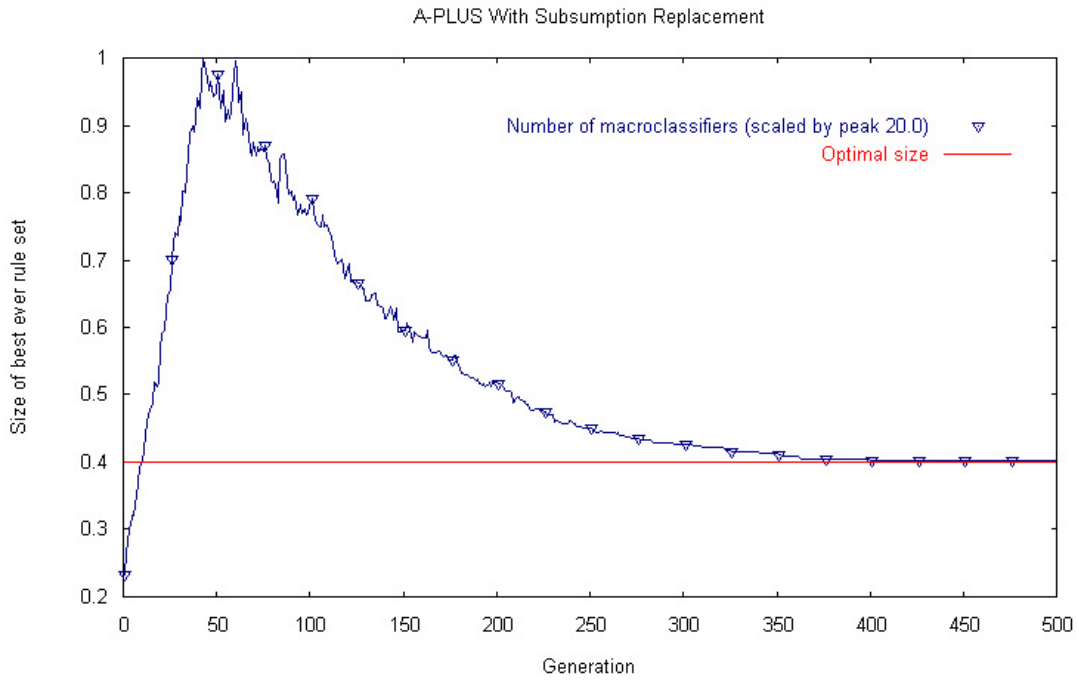
The graph below shows the fitness plots for the subsumption replacement experiment. The best and average fitness curves rise more steeply within the first 100 generations than the corresponding curves from the 'subsumption deletion' version. This increase in the average fitness could be due to the higher number of good classifiers now being preserved during crossover, leading to more stable rule sets. This version of the system took an average of 139.5 generations to find a rule set with optimal fitness, with the shortest time being an

impressive 48 generations and the longest being 266. This appears to be a considerable speed improvement over A-PLUS with subsumption deletion (average 203.8 generations, shortest 104, longest 380). The results of the paired t-test indicated that there was indeed a significant improvement in the best fitness data ($t = 3.98$, $t_{crit} = 2.35$, $p \leq 0.01$) and the average fitness data ($t = 3.20$, $t_{crit} = 2.35$, $p \leq 0.01$) when subsumption replacement was used, thus the null hypothesis was not upheld.



There was also a noticeable improvement in the *quality* of the solutions generated, which is an important factor in data mining tasks. This time 29 out of the 30 runs succeeded in finding the optimal concept description with eight macroclassifiers, whilst the remaining test run did find the eight desired rules but also had a single extra cheat rule. This suggests that the changes to the algorithm had the desired effect of increasing the influence of the correct rules and thus driving out undesirable classifiers.

The graph below shows the growth curve for the best ever rule set. The peak size was approximately the same as that for the previous version of the system, but the number of macroclassifiers now falls off more quickly from this peak, and stabilises at the optimal size after around 400 generations.



7.3.3 Conclusion

It has been found that the replacement, rather than deletion, of subsumed classifiers within the rule sets helps the system to filter out superfluous rules and find the maximally general macroclassifiers more consistently. In addition to the improved quality of the final solutions, statistical analysis has shown that there is a significant decrease in the time taken for the best and average fitnesses to reach their optimal values. This version of A-PLUS was extremely successful at learning the 6-multiplexer problem, finding the minimal length target concept description in almost 100% of the test runs. It should be noted that these results were obtained using a very simple Genetic Algorithm, highlighting the influence of the techniques used for evaluating and controlling the size of the rule sets.

Chapter 8

Breakdown of the Algorithm

8.1 Overview

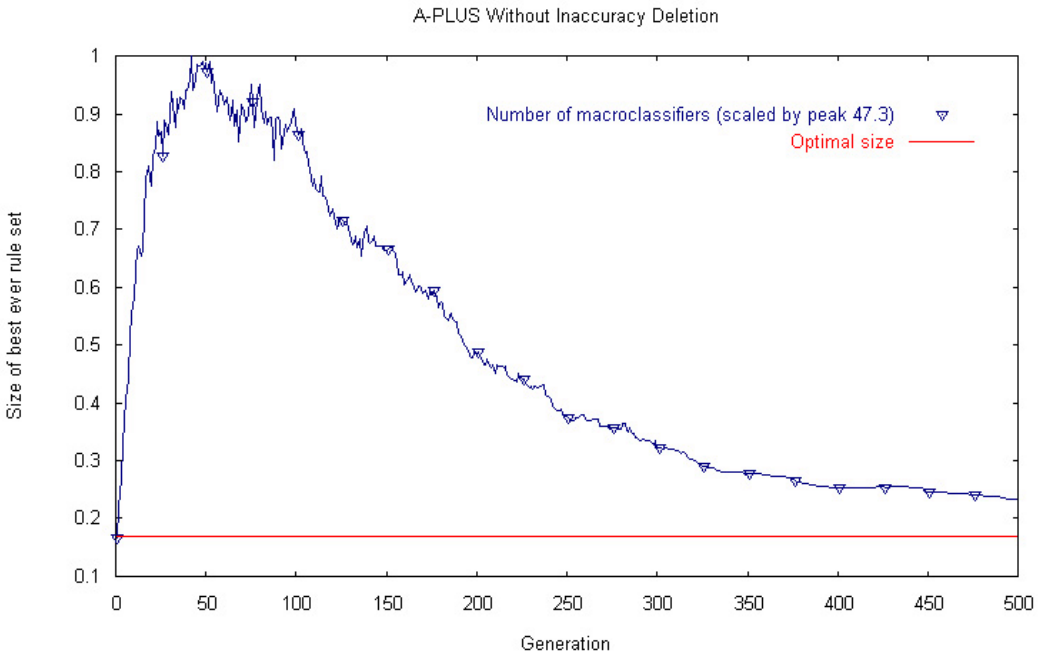
Having established that the new version of the system was capable of producing significantly better results than previous efforts, it was necessary to break down the algorithm in order to discover the exact effect of each new section. Further experiments were therefore carried out, removing different sections of the code – omitting inaccuracy deletion, omitting subsumption replacement, and finally omitting both of these (leaving only the niche-based fitness evaluation). It was found that subsumption of specific rules had the greatest effect with regard to bloat control, while the role of inaccuracy deletion was to focus the search for highly fit individuals more quickly.

8.2 Removal of Inaccuracy Deletion

In this experiment, we removed the section of code in which totally inaccurate classifiers, and below-average classifiers from large action sets, were deleted. As before, the system was run 30 times on the 6-multiplexer problem (using the same set of parameters), and the results were averaged to produce fitness and size curves. Observations were made regarding the quality of the solutions generated and the number of generations required to find a rule set with fitness 1.0.

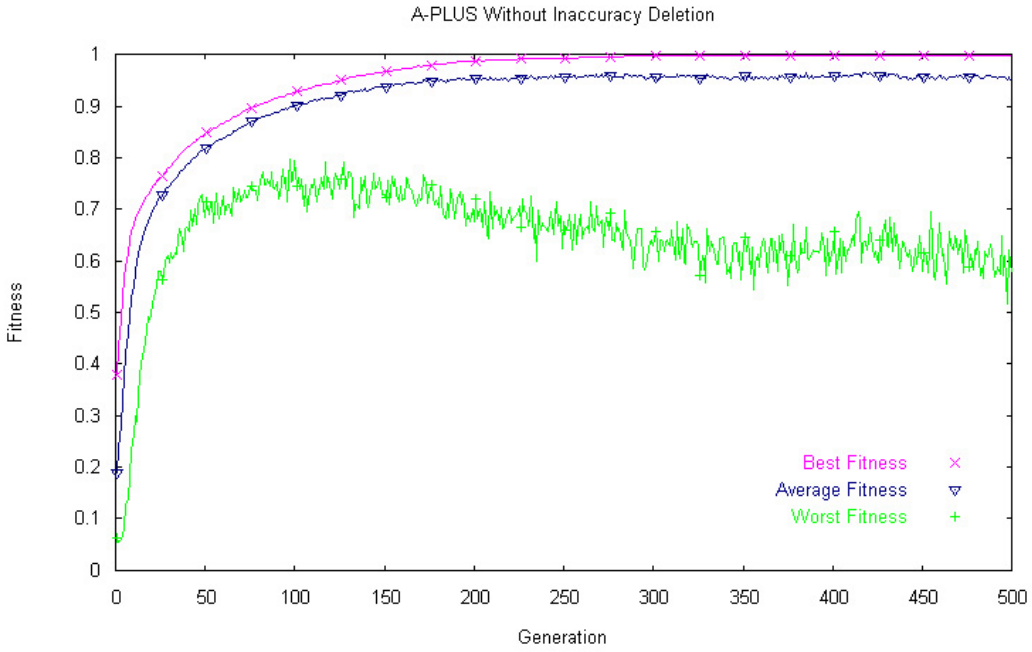
8.2.1 Results

The size curve for the best ever rule set shows that with subsumption replacement still in place, the number of macroclassifiers decreases to a reasonable level, although the peak size is more than twice that seen in the experiments when inaccuracy deletion was used. Only 5 out of 30 runs succeeded in finding the minimal length concept description, with every other run retaining a number of totally inaccurate macroclassifiers (as many as twelve in one case). 13 solutions also contained between one and five cheat rules, which is unsurprising as the 'good' classifiers would be less dominant in such a mixed rule set. Despite this, all but one solution did include the eight desired macroclassifiers (while the other found seven of them, plus one that was not maximally general). This shows that subsumption replacement enables the system to consistently find the correct rules even with inaccurate classifiers cluttering the rule set.



There was, however, an increase in the time taken for optimal rule sets to be found. The average time was 240.3 generations, the shortest 138 and the longest 493 – compare these figures with 139.5, 48 and 266 respectively for the full version of A-PLUS. This suggests that with inaccurate or inadequate classifiers present in the rule sets, the system has more difficulty in focusing its search and therefore takes longer to evolve an individual with the highest possible fitness.

Below are the fitness curves for this experiment, confirming that the rise in best and average fitness during the early stages is slightly less steep than when poor classifiers are deleted.

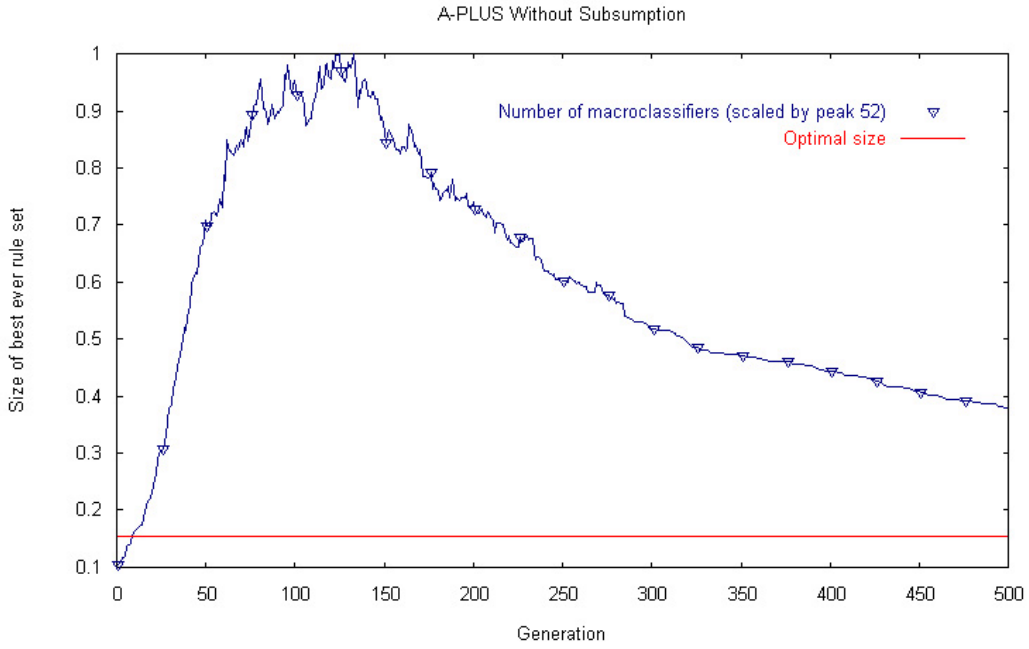


8.3 Removal of Subsumption

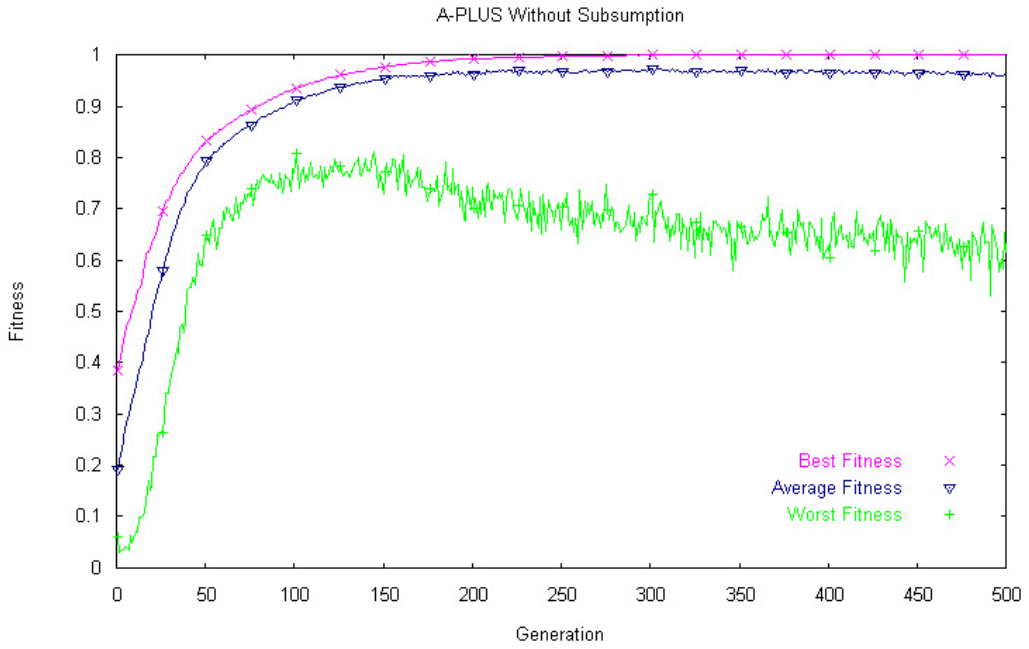
This time, inaccuracy deletion was left in, while the code for subsumption of too specific classifiers was commented out. The same experimental parameters were used as before.

8.3.1 Results

The significance of subsumption in bringing down the size of the best ever rule set became apparent after this experiment. With subsumption removed, the solutions were all far larger than necessary. No runs even succeeded in finding the eight ideal macroclassifiers; instead they all had non-maximally general classifiers (one run had as many as 25 of these), including some totally specific ones, i.e. copies of the data instances. 26 out of 30 solutions also included anywhere between one and ten cheat rules. The graph of solution growth confirms that the concept descriptions generated were well above the optimal size indicated by the flat line.



Whilst the removal of subsumption left the system struggling to find a concise description of the target concept, it appeared to have little effect on the speed with which perfect-scoring rule sets were found. Individuals with fitness 1.0 were found within 170.8 generations on average (shortest 86, longest 314), which is still respectable when compared to the full version of A-PLUS. The fitness curves are shown below.

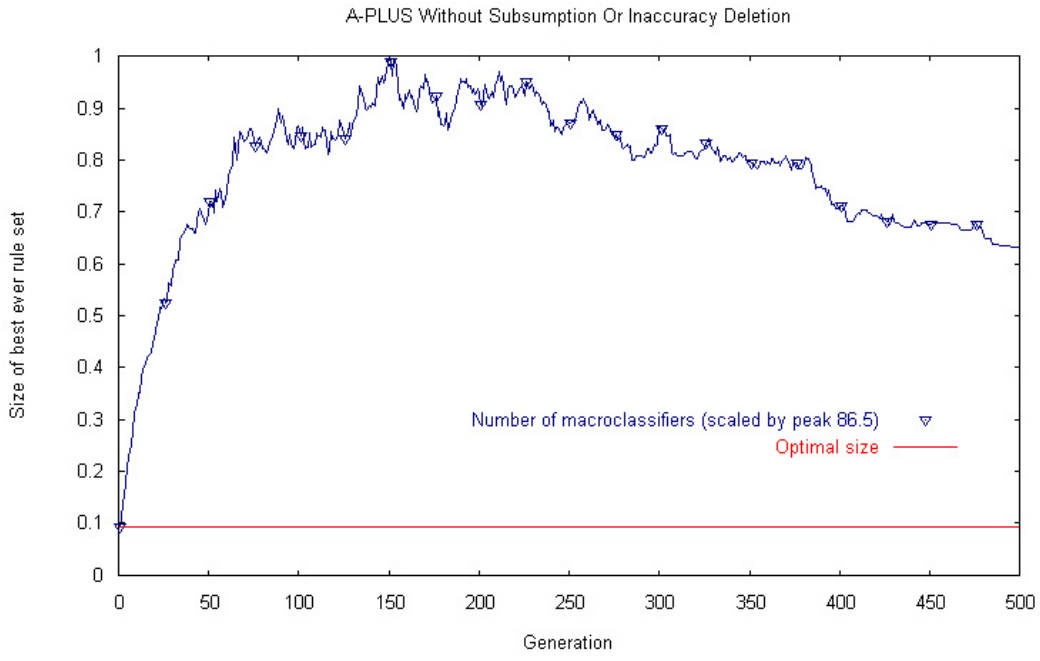


8.4 Removal of Inaccuracy Deletion and Subsumption

The final experiment saw the removal of both sections of code, in order to determine what effect the niche-based fitness evaluation alone had on the system's performance.

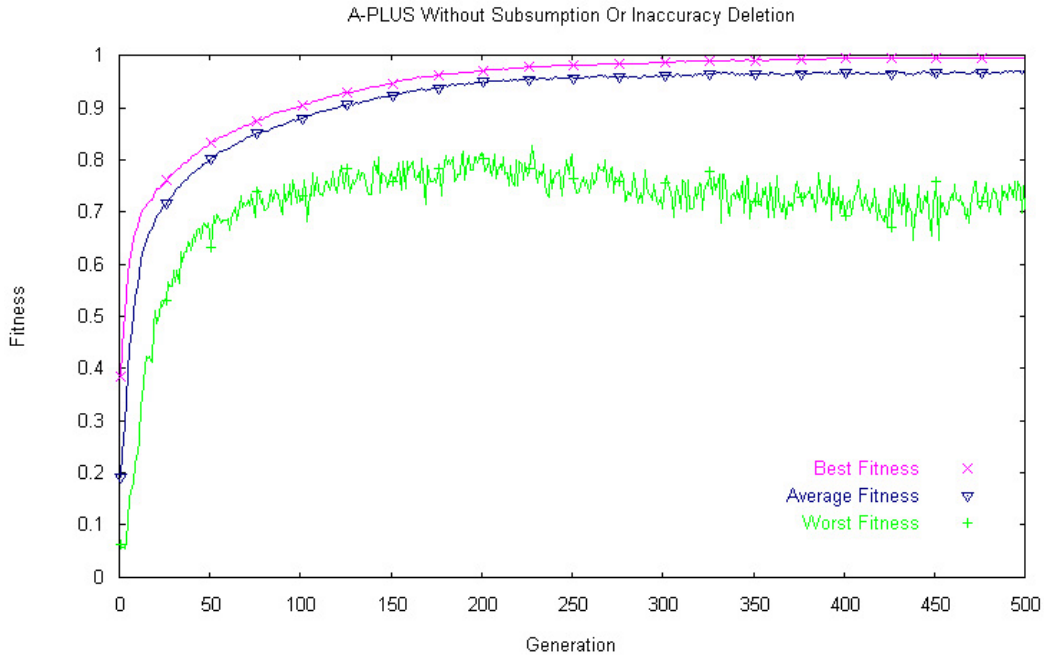
8.4.1 Results

Without the explicit removal of any classifiers from the rule set, the system came nowhere near to finding a short concept description, as can be seen from the graph below. The peak size of the best ever rule set was now around four times that seen in the full version tests, while the final solutions ranged in size from 20 to 100 macroclassifiers. Nevertheless, the solution can be seen to steadily decrease in size, indicating that the new fitness function does have some effect at controlling bloat.



Subsumption and inaccuracy deletion undoubtedly play a major role in guiding the search for good solutions. Without them, not a single run managed to find all eight ideal macroclassifiers, and 9 of the 30 runs did not find a perfect solution at all within 500 generations. Of the remaining 21 runs, the average time to find a rule set with optimal fitness was 382.4 generations, while the shortest was 215.

The fitness curves for the final experiment are presented below. A potentially misleading observation is that the worst individual tended to have a better fitness than in the full version of the system. While this could be interpreted as an improvement, the most likely explanation is that here, even the worst rule sets are able to obtain reasonable scores due to the presence of below-average classifiers which would normally have been deleted. It could be argued that the removal of classifiers therefore removes useful genetic material from the population; however, the impressive results obtained with the full version of A-PLUS indicate that there is enough genetic material present for the search even when rules are deleted.



8.5 Conclusion

The breakdown of the algorithm provided insight into the roles played by the different sections. It was found that subsumption is important for reducing the size of the final solutions, by removing large numbers of specific rules and replacing them with the appropriate generalisations. Deletion of inaccurate classifiers, meanwhile, helps the system to focus its search for optimal rule sets more quickly. The measurement of fitness within the niches prevented the system from suffering uncontrolled bloat, but the solutions were still extremely poor without the explicit pruning of unwanted classifiers.

Bacardit and Garrell (Bacardit 2003) also employ a rule deletion operator, removing from a rule set any classifiers that were not matched by any training instance, after computation of the individual's fitness. In order to avoid a loss of diversity early on, they recommend that the rule deletion operator is not activated until a predefined generation is reached. However, our deletion and replacement operators were used throughout the evolutionary run with no apparent ill effects. As with A-PLUS, Bacardit and Garrell combined rule deletion with hierarchical tournament selection in order to control bloat, although they used a very different fitness calculation. They also recommend the specification of a minimum number of classifiers to be left within a rule set; in our system this minimum was a single classifier.

Chapter 9

Further Testing

9.1 Overview

In order to explore the capabilities of A-PLUS more thoroughly, the system was tested on some further Data Mining tasks in addition to the 6-multiplexer problem. These tasks were the 11-multiplexer and the three Monk's problems; the various datasets are described in chapter 5. It was found that the system performed well on the relatively simple Monk's 1 problem, but had more difficulty with the harder problems. The search for more complex generalisations caused the bloat problem to reappear, which meant the system ran extremely slowly and it was almost impossible to obtain any useful results. It was therefore necessary to make some additional changes to the system. Adjusting the fitness calculation and placing a limit on the length of individuals helped to fix the problem, allowing the system to learn the 11-multiplexer concept within a reasonable amount of time. It was also necessary to impose a size limit for the Monk's 2 problem, whose target concept proved too challenging for A-PLUS, although a reasonable fitness curve was generated. Finally, Monk's 3 tested the system's ability to cope with noisy data. This did not present A-PLUS with any serious problems; several of the ideal macroclassifiers were successfully found.

9.2 Monk's 1

The first of the Monk's problems is not much more difficult to learn than the 6-multiplexer problem; the concept can be described with 12 macroclassifiers using Saxon and Barry's

binary encoding scheme. 30 test runs were carried out for this problem, and the resulting data averaged and plotted as before.

9.2.1 Parameters

The following system parameters were used for each run. Note that individuals are now initialised with 12 rules, since this is known to be the optimal length of the target concept description. An existing training set of 124 Monk's 1 data instances was used.

Data instance parameters:

condLength:	10
actLength:	1
numInstances:	124

Rule set parameters:

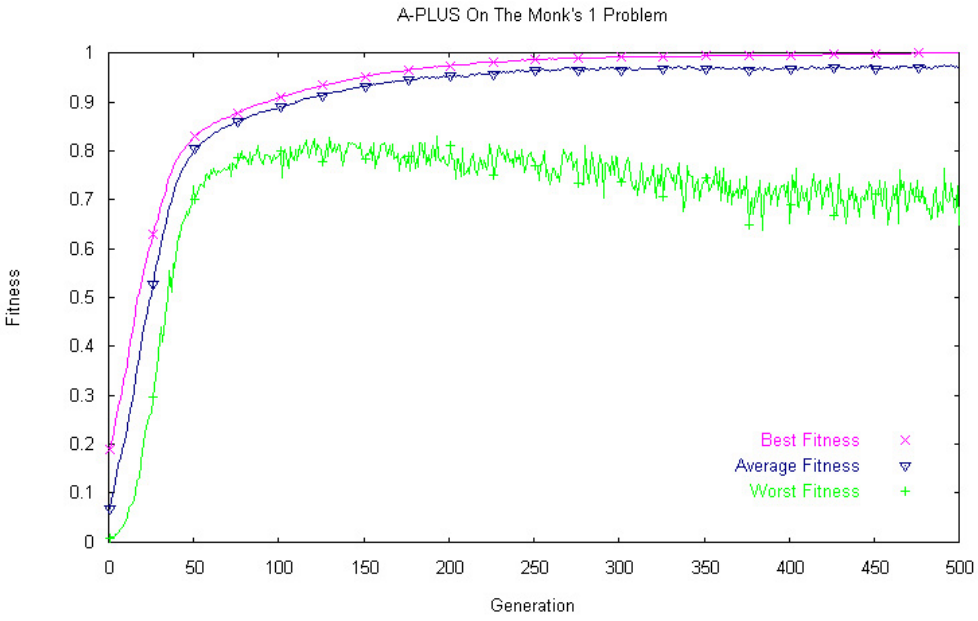
initialNumRules:	12
generality:	0.33
deletionProbability:	1.0
subsumptionProbability:	1.0

GA parameters:

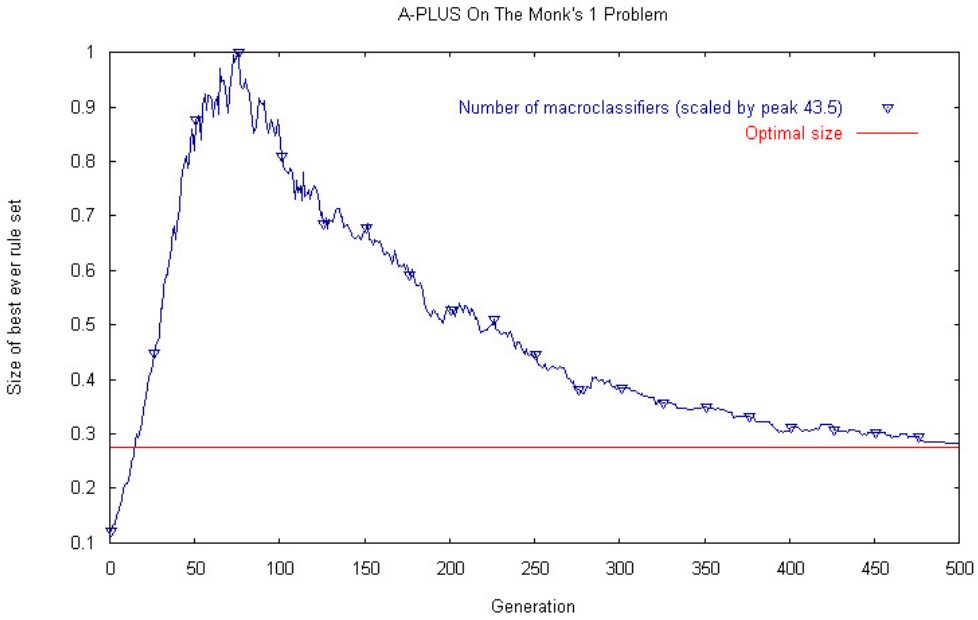
popSize:	50
crossoverRate:	0.6
mutationRate:	0.001
maxGenerations:	500
tournamentSize:	5
fitnessThreshold:	0.005

9.2.2 Results

The system succeeded in learning the concept without much difficulty. As both the concept description and the rules themselves are slightly longer for this problem than those in the 6-multiplexer problem, it seems reasonable to expect that the system would take longer to discover them. The results confirm this, with the average time to find an optimal rule set being 288.7 generations; the shortest time was 110 generations and the longest was 492.



Although only 10 out of 30 runs managed to find the twelve ideal macroclassifiers for this problem, the other 20 solutions were also quite good, with the vast majority finding at least ten of the ideal macroclassifiers and no runs finding less than eight of them. It could be that small adjustments to the system parameters would yield more consistent results, but time constraints prevented further experimentation. The best ever rule set shrank to the appropriate size in most of the test runs; only 3 solutions were larger than necessary, with the largest of these containing ten of the ideal macroclassifiers and six non-maximally general rules. The growth curve is shown below.



9.3 11-multiplexer

The basic idea behind this problem is the same as that of the 6-multiplexer. However, in this case the system must determine where to place wildcards within an 11-bit condition string, and it must find 16 macroclassifiers in total to describe the target concept. The problem is therefore considerably more difficult than the previous version, while the longer multiplexer problems have proved challenging for even some of the best concept learning systems.

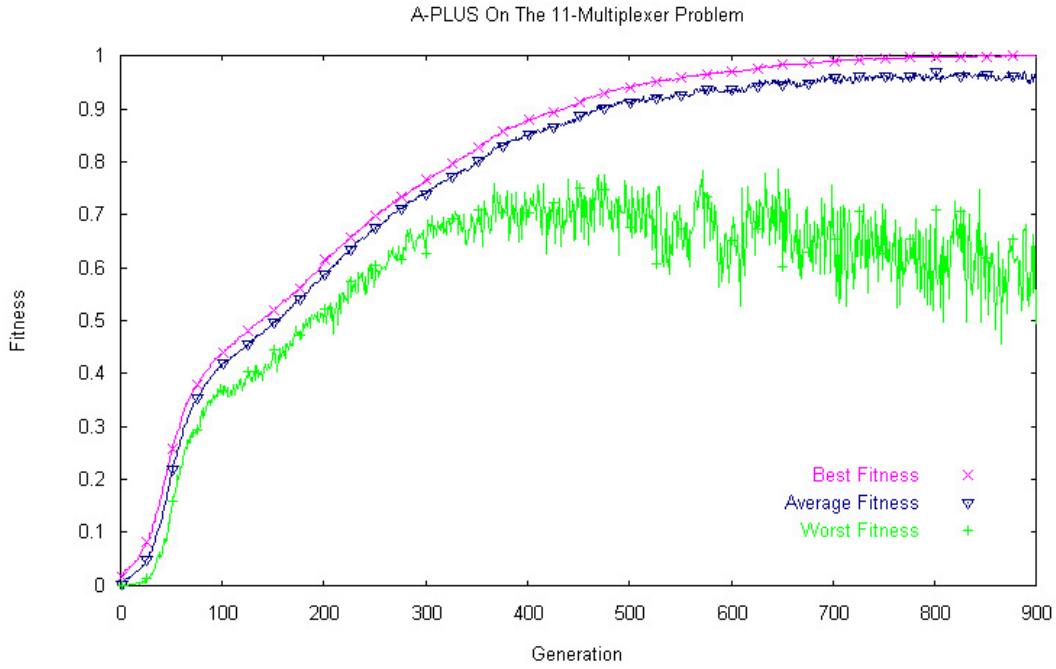
A-PLUS was tested on this problem using the same parameters as before, though in this case individuals were initialised with 16 rules, while *condLength* and *numInstances* were 11 and 2048 respectively. However, in searching for a set of rules to capture this more complex target concept, the system once again began to generate extremely long individuals which took increasingly longer to manipulate and evaluate. The process slowed to such a rate that testing became impractical, and since the adjustment of parameters such as *popSize* and *initialNumRules* proved ineffective, it was decided that some additional steps would need to be taken in order to control the bloat.

Examination of the population at a point where performance had become sluggish revealed that the individuals tended to be of very similar fitness, which would lead to uncertainty about where to focus the search. The payoff function was therefore adjusted to provide a clearer distinction between individuals; the fitness calculation was carried out in the same manner as before, but this time the final value was squared. This helped to speed up the search process somewhat, but did not remove the bloat effect, so it was also necessary to introduce an upper limit on the length of the individuals (as is often done within Genetic Programming). The crossover function was altered so that offspring would be rejected if their length exceeded a given limit, in which case the parent individuals would be placed directly into the new generation instead. Informal experiments suggested that 300 rules was a suitable size limit.

Since the 11-multiplexer concept takes longer to learn than the previously tried tasks, *maxGenerations* was increased to 900. A-PLUS still runs quite slowly for this problem, so it was only possible to carry out 10 test runs rather than the usual 30.

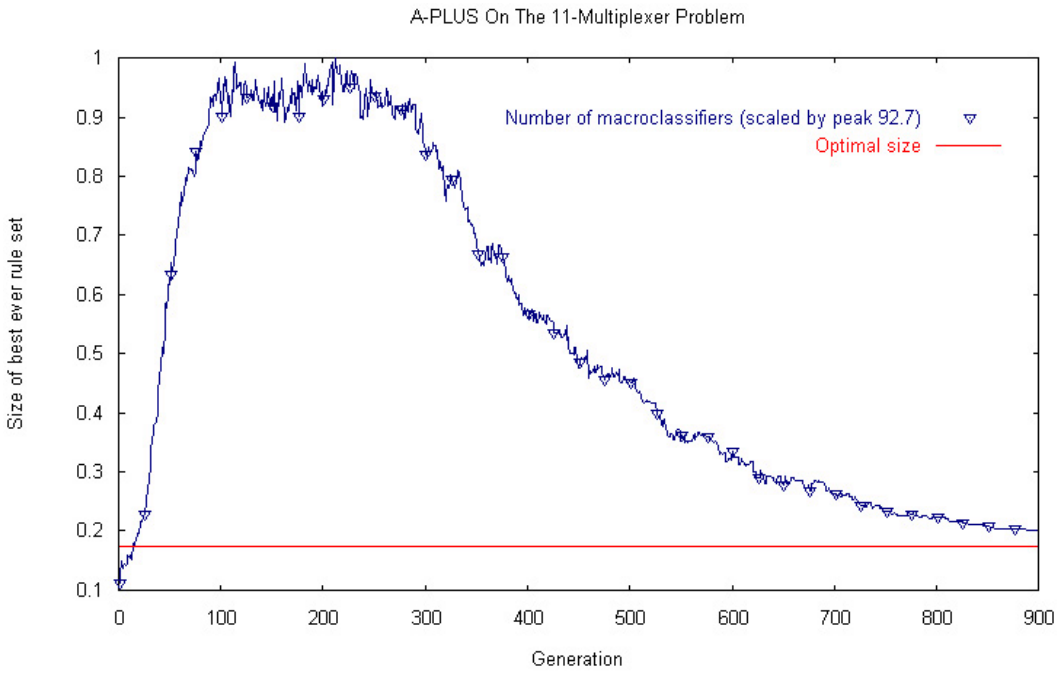
9.3.1 Results

The graph below shows the fitness curves for the 11-multiplexer problem. As one would expect, the best and average fitnesses rise much less steeply for this harder problem. On average, it took the system 608.8 generations to find a rule set with fitness 1.0, with the times ranging from 422 to 871 generations.



The quality of solutions was reasonably good. 8 of the 10 test runs succeeded in finding the sixteen ideal macroclassifiers, and the other 2 runs found fifteen of them, plus one rule that was not maximally general.

The average growth curve for the best ever rule set is shown below. 3 of the solutions shrank to the optimal size, while the others all included a number of cheat rules (ranging from one to eight of them). It is likely that the necessary limit imposed on the length of individuals had some effect on the quality of the final solutions; the better rules would not have been able to spread throughout the rule sets quite so much, and so these extra cheat rules were not always driven out. This demonstrates one weakness of using an arbitrary size limit as a form of bloat control.



It was thought that increasing the generality to 0.5 would also help to speed up the search process, since the final solution would need to include a large number of wildcards. However, it was found that this higher generality setting often caused premature convergence upon over-generalised rule sets, and even when this did not happen, the solutions did not appear to be significantly better than those obtained with a generality setting of 0.33.

9.4 Monk's 2

The target concept for the second of the Monk's problems is extremely challenging for the system to learn. There was an immediate problem with the sparse coding used; since not all binary values are legal, our random initialisation strategy meant that the vast majority of rules were totally inaccurate and were therefore deleted immediately. This left so little genetic material in the population that the selection process would often break down entirely due to all individuals having zero fitness, and at best there was rapid convergence upon extremely overgeneral rule sets. To avoid this problem, we instead initialised the rules sets by taking a random sample of data instances from the training set and adding wildcards according to the generality parameter. This is done in many other Machine Learning algorithms as a way of providing sufficient genetic material without introducing domain knowledge to the system.

The complex target concept caused problems for the system even when the GA was given plenty of material to work with; again a tendency for bloat was encountered, so a maximum length of 500 rules was imposed on the individuals. The system was run for 1000 generations to allow time for the best ever rule set to decrease in size.

9.4.1 Parameters

Data instance parameters:

condLength:	17
actLength:	1
numInstances:	169

Rule set parameters:

initialNumRules:	15
generality:	0.33
deletionProbability:	1.0
subsumptionProbability:	1.0

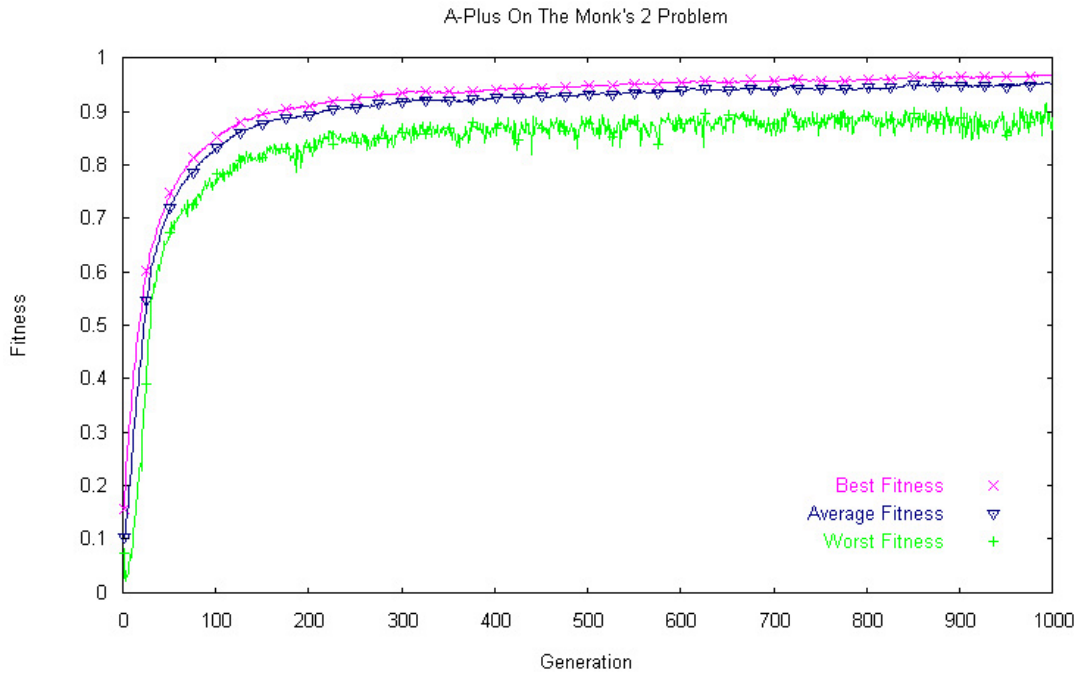
GA parameters:

popSize:	50
crossoverRate:	0.6
mutationRate:	0.001
maxGenerations:	1000
tournamentSize:	5
fitnessThreshold:	0.005

An existing training set of 169 data instances was used, with Saxon and Barry's enumeration encoding scheme.

9.4.2 Results

The fitness curves for this problem are shown below. Due to time constraints these results could only be averaged over 10 test runs.



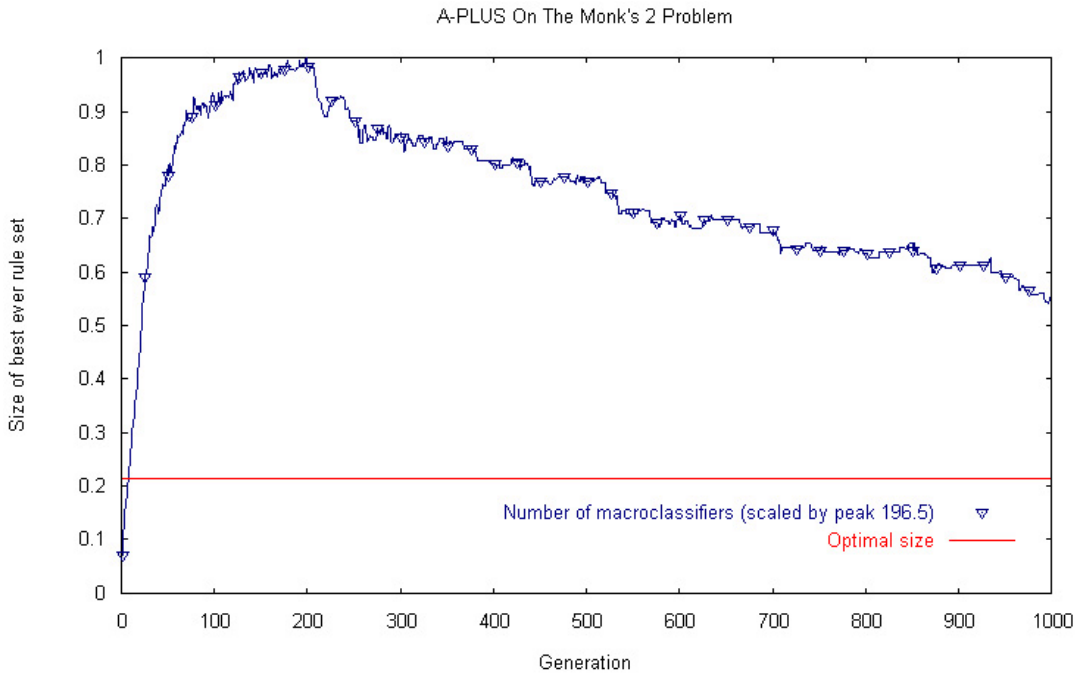
As can be seen in the graph, A-PLUS was unable to find a perfect rule set to describe the Monk's 2 problem; there is very little improvement in fitness after the first three or four hundred generations, suggesting that premature convergence may have occurred, although the fitness could still be increasing at a very gradual rate. A-PLUS clearly has difficulty in finding the correct generalisations, and a number of classifiers in the best ever rule sets contained slightly too many wildcards, making them only partially accurate. More detailed analysis of the generated solutions is both difficult and time-consuming due to the size of the rule sets and the number of ways in which the same attribute-value pairs may be represented.

Nonetheless, it is encouraging that the fitness of the best ever rule sets was around 0.97 on average; some improvement might be gained through experimentation with the system parameters and the maximum size of the individuals. The best solution had fitness 0.992 and successfully found eight of the ideal macroclassifiers.

It should be noted that in XCS (and in other fields such as Genetic Programming), a covering operator is employed throughout the evolutionary run. If a data instance does not match any of the existing rules, then a new classifier is created by adding some generality to the unmatched instance. Our system may therefore be at a disadvantage since we have only carried this out when generating the initial population for Monk's 2. It is difficult for A-PLUS to discover classifiers that are 100% accurate, which limits the amount of

subsumption that can occur, reducing the effectiveness of our bloat control techniques. It would be interesting to see if performance could be improved by continued use of covering.

Below is the growth curve for the Monk's 2 problem. The best ever rule set can be seen to decrease in size, although not enough for an optimally concise concept description; the final size of solutions ranged from 85 to 137 macroclassifiers (108.3 on average). However, the curve suggests that the best ever rule set may well continue to shrink if the system is run for longer than 1000 generations.



It was thought that a slight adjustment to the subsumption strategy might help to focus the search. Our usual technique is to subsume only classifiers that are 100% accurate. This means that even if a good generalisation is found, the influence of that classifier on the overall fitness value may be diluted due to the presence of several partially accurate classifiers in the same action set. We therefore repeated the experiments, this time allowing *all* unnecessarily specific classifiers to be subsumed regardless of their accuracy. However, there was no apparent improvement to either the fitness or growth curves. It is hard to understand the reason for this without analysing the solutions in greater detail, but it could be that our existing strategy for removing below-average classifiers from the action sets is effective enough to prevent this hypothesised dilution effect.

9.5 Monk's 3

The final problem on which A-PLUS was tested is the third of the Monk's problems. Although the target concept for this problem can be represented using ten macroclassifiers under the binary encoding scheme, the presence of noise in the data (that is, misclassified data instances) means that the system may not find this optimal description. Macroclassifiers which ought to be 100% accurate may appear not to be, and will therefore not be eligible to subsume more specific rules. This reduces the likelihood of the best classifiers dominating the rule set and driving out weaker ones. Similarly, macroclassifiers which should be totally inaccurate may seem to correctly classify one or more data instances, and will not be automatically deleted.

9.5.1 Parameters

The system did not suffer from bloat when learning this problem, so it was not necessary to impose any limit on the size of individuals. A training set of 122 data instances was used with Saxon and Barry's binary encoding scheme, and 30 tests runs were carried out.

Data instance parameters:

condLength:	10
actLength:	1
numInstances:	122

Rule set parameters:

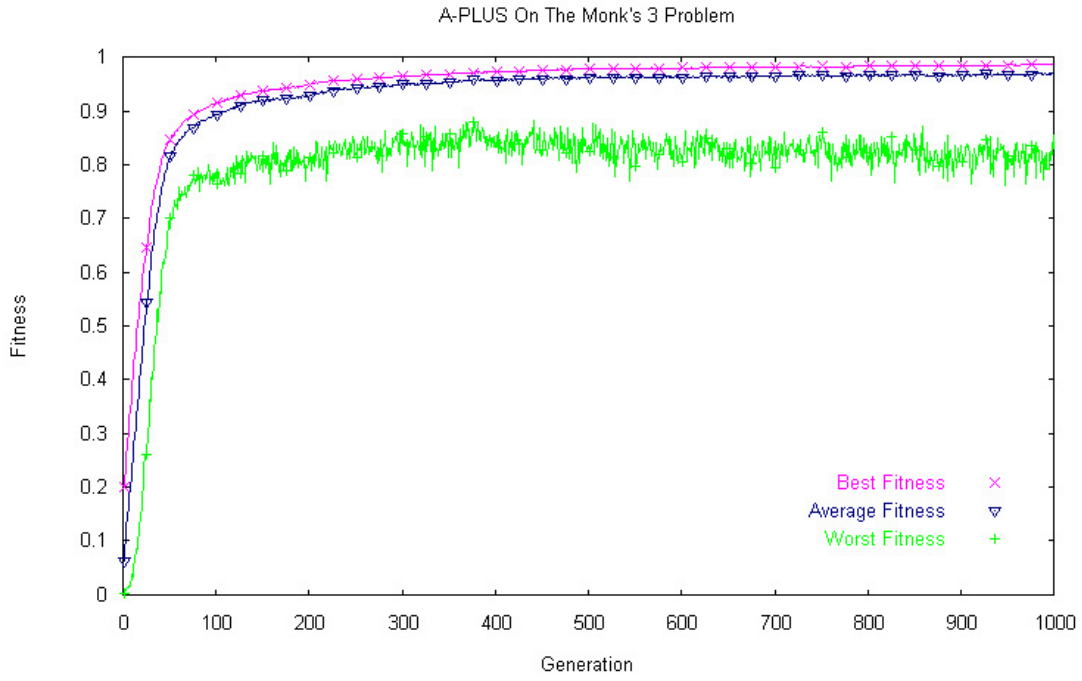
initialNumRules:	10
generality:	0.33
deletionProbability:	1.0
subsumptionProbability:	1.0

GA parameters:

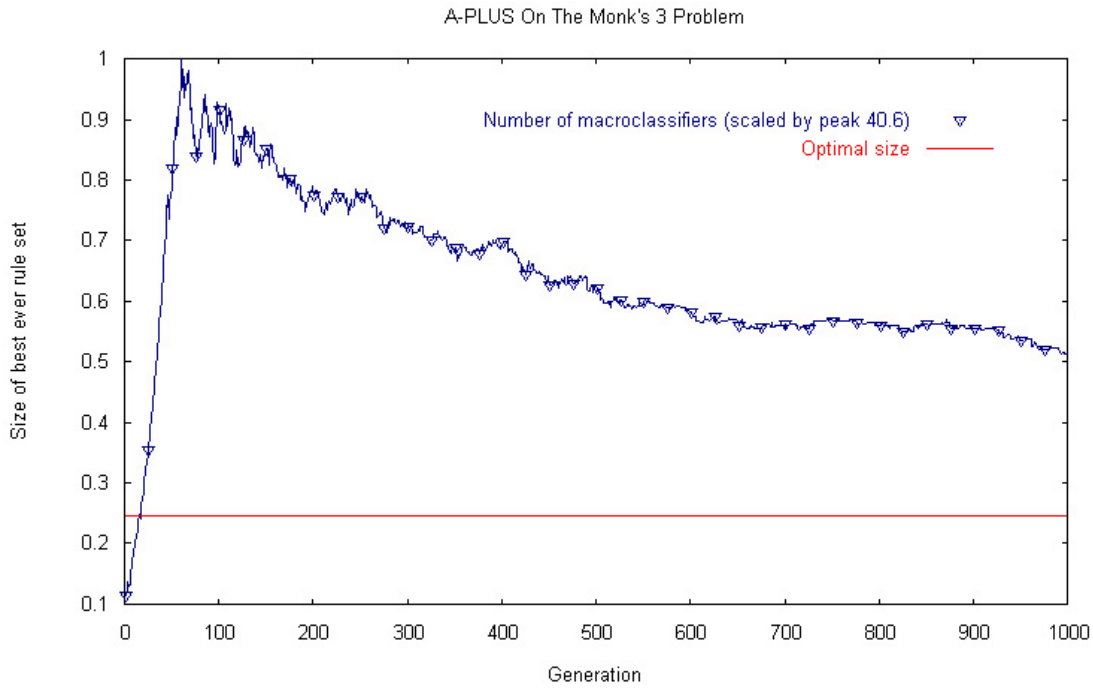
popSize:	50
crossoverRate:	0.6
mutationRate:	0.001
maxGenerations:	1000
tournamentSize:	5
fitnessThreshold:	0.005

9.5.2 Results

The solutions generated in response to this problem were as one might expect; good, due to the straightforward target concept, but not perfect due to the presence of noise in the data. Most solutions contained three or four of the ideal macroclassifiers, along with many non-maximally general versions of other accurate rules. Certain classifiers proved easier to find than others; the rules #0####1###:0, ####1#1###:0 and ###1##1###:0 appeared in almost every solution, while the other rules were found with varying frequency and not always in their most general form. Additionally, 20 out of 30 runs retained the rule #####11###:0, which does not belong to the target concept description. Examination of the training set diagram in (Thrun 1991) provides an explanation for this; the positive examples that match this classifier are not included among the training instances, thus the system believes this rule to be 100% accurate.



The vast majority of solutions had a fitness of at least 0.98, with two achieving optimal fitness, presumably by chance coverage of the entire training set including misclassified instances. The lowest fitness seen among the best ever rule sets was just over 0.96, which is highly respectable; one would expect the fitness not to exceed 0.95 due to the 5% noise in the data. This high level of accuracy suggests that A-PLUS is not over-fitting to the training data (that is, being 'fooled' by the noise and over-generalising), although further investigation would be required to confirm this.



The average size of the solutions was 20.9 macroclassifiers – approximately twice the size of the optimal concept description. This is mainly due to the number of non-maximally general rules that were included, which would all have been subsumed if the system were able to recognise an optimal generalisation. The fact that bloat did not occur and the solutions were of a reasonable size indicates that the noise in the data was not especially confusing to the system.

Chapter 10

Conclusion

10.1 Evaluation and Future Work

This dissertation has described the main problem with existing Pittsburgh Learning Classifier Systems, namely their relative inefficiency, and investigated a number of techniques for improving their performance. The main focus of our experiments has been on controlling the bloat effect, the phenomenon in which fitness-based selection causes rule sets to grow exponentially, slowing and reducing the effectiveness of the search. We have incorporated several ideas from Wilson's Michigan approach system XCS (Wilson 1995) into a simple Pittsburgh LCS and analysed their effects.

Firstly, we introduced a payoff function where fitness is measured within the various niches of the environment, thus penalising classifiers which advocate the same action in situations where it is inappropriate. This gave a small improvement in the system's performance, with the best ever rule set gradually decreasing in size instead of continuing to grow out of control. We also introduced the explicit pruning of various kinds of 'weak' classifiers. Totally inaccurate classifiers are removed automatically, and classifiers appearing in large action sets are removed with a specified probability if their accuracy is below the average within that action set. It was found that this helped to focus the search, allowing rule sets of optimal fitness to be found more quickly. Finally, accurate classifiers are removed according to a specified probability if there is another accurate classifier that subsumes them, i.e. a rule that is more general. This was shown to significantly reduce the size of the solutions generated by the system, often resulting in the optimally concise concept description being found.

Our system, known as A-PLUS, was tested on five standard data mining tasks: the 6-multiplexer and 11-multiplexer problems, and the three Monk's problems. The target concepts for these five problems vary greatly in complexity, giving us a feel for the extent of the new system's capabilities. It was found that A-PLUS performed extremely well on relatively simple concepts such as the 6-multiplexer and Monk's 1 problem. The system took an average of 139.5 generations to find an optimal rule set for the 6-mux problem, and even managed this within as few as 48 generations. This is impressive when compared to results obtained with XCS, which takes around 3,000 to 3,500 generations to achieve 100% accuracy when using 400 microclassifiers (Barry 2000). When making this comparison it is important to remember that the two systems use very different approaches, but nevertheless, the small number of generations required with A-PLUS is certainly appealing. Similarly, XCS takes around 10,000 generations to learn the Monk's 1 problem (Saxon 1999), whereas A-PLUS took an average of 288.7 generations with the shortest number being 110. It is worth noting that unlike XCS, our system does not employ any covering mechanisms for dealing with unmatched instances.

Another advantage of A-PLUS is that it is essentially simpler in construction than XCS, with arguably a more natural balance of pressures. Larry Bull points out (Bull 2003) that "whilst XCS has been shown to be extremely effective in a number of domains, its complexity can make it difficult to establish clear reasons for its behaviour". Bull has created a much simplified version known as YCS, without a triggered niche GA or any form of subsumption, in order to better understand the effects of accuracy-based fitness in general. However, the relative simplicity of a Pittsburgh LCS and the small number of parameters required make it far less complex than a Michigan-style system, even YCS. There has been a great deal of research into understanding how the GA works, and this analysis can be applied directly to the Pittsburgh LCS in order to give similar improvements. Additionally, Pittsburgh systems offer the potential for further speedups via parallelisation. A useful starting point would be to compare the performance of A-PLUS with results obtained using other Pittsburgh approach systems, most notably GABIL (De Jong 1993).

A-PLUS had more difficulty with harder problems such as the 11-multiplexer and Monk's 2, where the rules are longer and the target concepts more complex. In these situations the mechanisms inspired by XCS proved insufficient to prevent bloat, and it was necessary to take additional steps. In order to solve the 11-multiplexer problem in reasonable time, fitness values were squared to provide clearer focus for the search, and a maximum length of 300 rules was imposed on the individuals. The system was then able to find an optimal

rule set within an average of 608.8 generations (where the longest time taken among the test runs was 871 generations). This is compared to the 12,000 generations required by XCS using a population of 800 microclassifiers (Barry 2000). The problem which proved the most challenging for A-PLUS was Monk's 2, having a very complex target concept. Firstly, the sparse coding for the problem meant that not all binary values were legal, and this resulted in the vast majority of randomly initialised rules being deleted due to total inaccuracy, leaving very little for the GA to work with. This was resolved by initialising the rule sets with randomly generalised versions of data instances from the training set, which is standard practice in many Machine Learning algorithms and provides sufficient genetic material without introducing domain knowledge to the system. It was again necessary to impose a size limit on individuals (this time 500 rules) to deal with the bloat effect. Although the system managed to find rule sets of good fitness (around 0.97 on average), it was unable to learn the target concept perfectly, and the solutions were somewhat larger than necessary. Further investigation is clearly required into how the performance of A-PLUS can be improved for difficult problems like Monk's 2; it may be helpful to use a more sophisticated Genetic Algorithm than the one employed in our experiments.

The Monk's 3 problem demonstrated that A-PLUS is not adversely affected by noise within the training data, although it would be interesting to see how well the system copes with data from real-world domains, such as that obtainable from the UCI Machine Learning Repository (Blake 1998). It may well be necessary to reduce the deletion and subsumption probabilities for some of these problems. A comparison of XCS with seven other learning algorithms on a range of datasets is available in (Bernadó 2002), so it is worth testing A-PLUS on the same problems to see how it compares. This will involve taking the best solution generated over the training set and using it to obtain a score over the test set of data instances. A-PLUS has been designed only for 'offline' learning tasks, where a set of pre-classified data instances is available; future work may involve creating an equivalent system for online learning such as solving 'animat' problems. It should be noted that accuracy is not necessarily a good fitness measure for all Machine Learning problems.

Although the results obtained with A-PLUS have largely been promising and we have taken steps to remove unnecessary repeated computation, there are a number of obvious improvements which could be made to the system. It is particularly inefficient to maintain a complete and unordered set of microclassifiers alongside the set of macroclassifiers, since the latter must be rebuilt every time the rule set undergoes crossover or mutation, or when

rules have been deleted due to inaccuracy. The more distinct rules there are present, the slower this updating process becomes, which can be a problem when the target concept has a complex description. A possible alternative approach would be for each macroclassifier to simply have an associated array of its copies' positions within the rule set. Crossover would then involve merging the position arrays of the affected macroclassifiers – somewhat reminiscent of Messy GAs (Goldberg 1989b), where the names of genes are transferred along with their values, and there are strategies for dealing with any overspecified or underspecified chromosomes that are produced. Another idea is to use only macroclassifiers but use the numerosity values to simulate a block of adjacent copies; this may prevent sufficient mixing of genetic material, however.

For the purposes of simplicity in this project, a Genetic Algorithm was effectively written "from scratch", incorporating only the bare functionality required. This made it easier to analyse the effects of the algorithmic changes we made to the LCS, and still allowed us to obtain respectable results in the various experiments. However, it is likely that much better performance could be achieved if a more complex existing GA was used, as the iterative nature of these algorithms makes them notoriously sensitive to subtle changes. An important next step would therefore be to take advantage of current GA research in order to fine-tune the search process more carefully, as well as experimenting more extensively with different system parameter settings. We have also acknowledged that Java programs are relatively slow, so it may be helpful to create an implementation of A-PLUS in a compiled language such as C.

Finally, there are a number of additional techniques which we did not have time to investigate here. We have yet to try integrating Grammatical Evolution within the Pittsburgh LCS; this may help to solve to the efficiency problems encountered with Monk's 2 and the 11-multiplexer, for example. Another suggestion is to use a form of elitism to ensure that the best individuals survive from one generation to the next; when carrying out crossover, the offspring could be evaluated immediately, and the two fittest individuals out of the four (parents and children) would be placed into the new population. Retaining the best genetic material in this way may lead to good solutions being found more quickly.

We are also interested in investigating the use of GA pressure – in addition to subsumption replacement – to reduce the size of rule sets. Once a rule set with fitness of 1.0 is found, the fitness calculation could include an added proportion: (number of accurate macroclassifiers) / (number of microclassifiers). This measure should mean that once a rule

set achieves 100% accuracy, pressure is applied to reduce the size of the rule set, driving out the more specific accurate classifiers. This may or may not be effective, since it is a generic pressure rather than one aimed at particular classifier groups within an individual.

Perhaps the most promising idea for bloat control is the use of "multi-objective methods". An apparent problem with many existing Learning Classifier Systems is that learning is treated as a single-objective task, when really it is an optimisation task with multiple objectives. Goldberg points out that even if a canonical 'simple LCS' could be agreed upon, the model would be far more complex than the 'simple GA', since an LCS tries to find "the (1) smallest set of rules that (2) best solves the example problem while (3) generalising well to all similar problem instances" (Goldberg 1992). Whilst the Pittsburgh approach has less conflicting objectives than a Michigan LCS, since the classifiers within an individual cooperate without competition, it is still true that there are multiple objectives to consider. De Jong et al present a similar argument in the context of Genetic Programming, suggesting that since the prevention of bloat and the promotion of diversity "are often implicit goals" in basic GP, "a straightforward idea is to make them explicit by adding corresponding objectives" (De Jong 2001). Traditional methods for coping with multiple objectives have involved introducing weights for linear combinations of the attribute values, or turning objectives into constraints with associated thresholds and penalty functions. These approaches can be problematic since the GA solutions are usually very sensitive to small changes in the penalty or weighting factors.

Horn et al explain that "the Genetic Algorithm, however, is readily modified to deal with multiple objectives by incorporating the concept of Pareto domination in its selection operator, and applying a niching pressure to spread its population out along the Pareto optimal tradeoff surface" (Horn 1994). This means using the GA to find all possible tradeoffs among the various conflicting objectives; the 'non-dominated' solutions (i.e. those for which no other individuals are superior in all attributes) lie along a surface in attribute space known as the *Pareto front*, and the goal is to find a representative sampling of solutions along this frontier. Maintaining diversity along the front should in theory help the search for new and improved tradeoffs, since parents from different parts of the Pareto front may generate offspring lying somewhere between them.

De Jong et al summarise the approach as follows: "The basic idea is to search for multiple solutions, each of which satisfy the different objectives to different degrees. Thus, the selection of the final solution with a particular combination of objective values is postponed until a time when it is known what combinations exist." They report that a

multi-objective approach to bloat within Genetic Programming "appears promising" and has been used before with fitness and size as objectives, but can result in premature convergence to small individuals. In (De Jong 2001), they experiment with using fitness, size and diversity as three explicit objectives of a search; their algorithm selects individuals if and only if they are not dominated by other individuals in the population, causing exploration to be focused by the objectives. The inclusion of the diversity objective allows them to achieve strongly positive results, keeping the visited trees small without requiring any size limit or relative weighting of fitness and size.

It would therefore be very interesting to incorporate similar ideas within a Pittsburgh Learning Classifier System, making desired characteristics of the search – such as compact solutions – into explicit objectives and selecting only non-dominated individuals from each generation. Some of these ideas have already been partially implemented in A-PLUS, as our subsumption strategy involves selecting a non-dominated classifier within a match-set niche. The results achieved indicate that the approach merits further investigation; a possible next experiment would be to use only these non-dominated classifiers when selecting material for the next generation, although this may result in an unacceptable loss of diversity.

10.2 Personal Reflection

On the whole, the project has been very successful. The use of accuracy-based fitness, subsumption and deletion as a form of bloat control had not previously been tried, and our experimental results have been encouraging, although there is clearly still room for improvement. It is rewarding to have made a significant contribution towards an active area of research.

The project has been a steep learning curve in many ways. Since we have tried to draw together ideas from several different areas of research, it has involved a large amount of background reading in order to gain a sufficient understanding of the various topics. Some of these topics are complex, and it has been necessary to acquire the skill of extracting relevant material whilst recognising which parts are beyond the scope of the current investigation. Time constraints prevented more thorough exploration of some ideas; in particular it was not possible to carry out detailed analysis of the problems encountered by A-PLUS on the harder data mining tasks, which makes it unclear which of our proposed

solutions would most likely be effective. It is hoped that the suggestions made in the dissertation will be followed up by other researchers in the field so that further progress can be made.

It was also challenging to design and implement a program as large and complex as a Learning Classifier System without having extensive programming experience. This was approached by choosing a familiar programming language that is relatively easy to debug, and building the system up gradually with a bare acceptable level of functionality. The resulting program has been sufficient to generate a set of useful results, since in this project the *comparison* of different algorithms was more important than the overall level of performance. Nonetheless, if more time were available or the project was to be repeated, a more detailed examination of existing Pittsburgh classifier systems would be carried out, and pre-written GA code would be integrated into the system to give better results. Improved C coding abilities would also be beneficial since a faster compiled language is arguably more suitable for the implementation of such a highly iterative system.

This project has involved the application of a number of key skills and knowledge acquired during the degree course, such as project planning, research, time management, appropriate tool selection, programming, scientific analysis and written communication. Despite the amount of work involved, the project has been interesting and much has been achieved within a relatively short space of time.

References

Bacardit, J., Garrell, J.M. (2003). Bloat control and generalization pressure using the minimum description length principle for a Pittsburgh approach Learning Classifier System. *Sixth International Workshop on Learning Classifier Systems (IWLCS-2003)*, Chicago, July 2003.

Banzhaf, W., Langdon, W.B. (2002). Some Considerations on the Reason for Bloat. *Genetic Programming and Evolvable Machines*, **3**(1), pp. 81-91.

Barry, A.M. (2000). *XCS Performance and Population Structure in Multiple-Step Environments*. Ph.D. thesis, Queens University Belfast.

Bassett, J. K. (2002). *A Study of Generalization Techniques in Evolutionary Rule Learning*. M.Sc. thesis, George Mason University, Fairfax, VA.

Bernadó, E., Llorà, X., Garrell, J.M. (2002). XCS and GALE: a Comparative Study of Two Learning Classifier Systems with Six Other Learning Algorithms on Classification Tasks. *Proceedings of the 4th International Workshop on Learning Classifier Systems (IWLCS-2001)*, pp. 337-341.

Blake, C.L., Merz, C.J. (1998). *UCI Repository of machine learning databases* [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science.

Bull, L. (2003). *A Simple Accuracy-based Learning Classifier System*. University of the West of England, Bristol. *Technical Report: UWELCSG03-005*.

Darwin, C.R. (1859). *The Origin of Species*.

De Jong, E.D., Watson, R.A., Pollack, J.B. (2001). Reducing Bloat and Promoting Diversity using Multi-Objective Methods. **In:** *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pp. 11-18. San Francisco, CA: Morgan Kaufmann.

De Jong, K.A., Spears, W.M. (1991). Learning Concept Classification Rules Using Genetic Algorithms. **In:** *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pp. 651-656. Sydney, Australia: Morgan Kaufmann.

De Jong, K.A., Spears, W.M., Gordon, D.F. (1993). Using Genetic Algorithms for Concept Learning. *Machine Learning*, **13**(2/3), pp. 161-188.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*, pp. 1-14. Reading, MA: Addison Wesley.

Goldberg, D.E., Korb, B., Deb, K. (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, **3**(5), pp. 493-530.

Goldberg, D.E., Horn, J., Deb, K. (1992). What Makes a Problem Hard for a Classifier System? **In:** *Collected Abstracts for the First International Workshop on Learning Classifier Systems (IWLCS-92)* (R.E. Smith, ed.), Houston, TX, October 1992.

Griffith, R. (2002). *A Java Implementation of Grammatical Evolution*. M.Sc. thesis, University of Bath.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: The University of Michigan Press.

Holland, J.H., Reitman, J.S. (1978). Cognitive systems based on adaptive algorithms. **In:** *Pattern-Directed Inference Systems* (D.A. Waterman, F. Hayes-Roth, eds.), pp. 313-329. New York: Academic Press.

Horn, J., Nafpliotis, N., Goldberg, D.E. (1994). A Niched Pareto Genetic Algorithm for Multiobjective Optimization. *Proceedings of the First IEEE Conference on Evolutionary Computation*, pp. 82-87.

Kodratoff, Y. (1988). *Introduction to Machine Learning*, pp. 138-139. London: Pitman.

Langdon, W.B. (1997). *Fitness Causes Bloat in Variable Size Representations*. University of Birmingham. *Technical Report: CSRP-97-14*.

Langdon, W.B., Poli, R. (1997). Fitness Causes Bloat. *Second On-Line World Conference on Soft Computing in Engineering Design and Manufacturing*, pp. 13-22. London: Springer-Verlag.

Langdon, W.B., Poli, R. (1998). Genetic Programming Bloat with Dynamic Fitness. *Lecture Notes in Computer Science: Proceedings of the First European Workshop on Genetic Programming* (W. Banzhaf, R. Poli, M. Schoenauer, T.C. Fogarty, eds.), **1391**, pp. 97-112. London: Springer-Verlag.

Llorà, X., Garrell, J.M. (1999). GENIFER: A Nearest Neighbour based Classifier System using GA. **In:** *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO99)*, p. 797. San Francisco, CA: Morgan Kaufmann.

Luke, S. (2000). *Evolutionary Computation*. [Online] Available at http://www.cs.umd.edu/users/seanl/gp/ (accessed February 2004).

Luke, S., Panait, L. (2002). Fighting Bloat with Nonparametric Parsimony Pressure. **In:** *Parallel Problem Solving from Nature* (J.-J. Merelo Guervós, P. Adamidis, H.-G. Beyer, J.-L. Fernández-Villacañas Martín, H.-P. Schwefel, eds.), pp. 411-420. Berlin: Springer.

Matsumoto, M., Nishimura, T. (1997). *Mersenne Twister: A random number generator*. [Online] Available at http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html (accessed February 2004).

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*, pp. 166-167. Cambridge, MA: The MIT Press.

Mitchell, M., Forrest, S. (1997). Fitness Landscapes: Royal Road Functions. **In:** *Handbook of Evolutionary Computation* (T. Bäck, D.B. Fogel, Z. Michalewicz, eds.), pp. B.2.7.5:1-25. New York and Bristol: Oxford University Press and Institute of Physics Publishing.

Ryan, C., Collins, J.J., O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Lecture Notes in Computer Science: Proceedings of the First*

European Workshop on Genetic Programming (W. Banzhaf, R. Poli, M. Schoenauer, T.C. Fogarty, eds.), **1391**, pp. 83-95. London: Springer-Verlag.

Ryan, C., O'Neill, M., Collins, J.J. (1998). Grammatical Evolution: Solving Trigonometric Identities. *Proceedings of Mendel 1998: 4th International Mendel Conference on Genetic Algorithms, Optimisation Problems, Fuzzy Logic, Neural Networks, Rough Sets*, pp. 111-119.

Ryan, C., O'Neill, M. (1998). Grammatical Evolution: A Steady State Approach. *Proceedings of the Second International Workshop on Frontiers in Evolutionary Algorithms*, pp. 419-423.

Saxon, S., Barry, A.M. (1999). XCS and the Monk's Problems. **In:** *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program* (A.S. Wu, ed.), pp. 272-281.

Smith, S.F. (1980). *A learning system based on genetic adaptive algorithms*. Ph.D. thesis, University of Pittsburgh.

Spears, W.M., De Jong, K.A. (1992). Using Genetic Algorithms for Supervised Concept Learning. **In:** *Artificial Intelligence Methods and Applications* (N.G. Bourbakis, ed.). Singapore: World Scientific.

Thrun, S.B., Bala, J., Bloedorn, E., Bratko, I., Cestnik, B., Cheng, J., De Jong, K., Dzeroski, S., Fahlman, S.E., Fisher, D., Hamann, R., Kaufman, K., Keller, S., Kononenko, I., Kreuziger, J., Michalski, R.S., Mitchell, T., Pachowicz, P., Reich, Y., Vafaie, H., Van de Welde, W., Wenzel, W., Wnek, J., Zhang, J. (1991). *The MONK's Problems: A Performance Comparison of Different Learning Algorithms*. Carnegie Mellon University. *Technical Report: CMU-CS-91-197*.

Wilson, S.W. (1995). Classifier Fitness Based on Accuracy. *Evolutionary Computation*, **3**(2), pp. 149-175.

Appendix – The Code

The code for the most important classes is given below. The classes included here are LCS, Population, RuleSet, Classifier, MacroClassifier, Instance and GeneticOps. Full code is available on the attached CD.

LCS

```
/* The top level control program. */

public class LCS {

    public static void main(String[] args) {

        // Initialise the output writers
        OutputFiles outFiles = OutputFiles.getWriters();

        // Read in the instances and initialise the GA population
        Population thePopulation = new Population();

        // Run the GA until the generation limit is reached
        thePopulation.adapt();

        // Close the output files
        outFiles.close();

    } //end main()

} //end class LCS
```

Population

```
import java.io.*;

/* This includes the set of instances, the GA population and a copy of the best rule
   set found. This class also contains the methods needed to evaluate the rule sets
   and run the GA. */

public class Population {

    private static boolean instantiated = false;

    private Parameters params = Parameters.getParams();
    private RandomNumbers generator = RandomNumbers.getGenerator();
    private GeneticOps genops = new GeneticOps();

    private OutputFiles outFiles = OutputFiles.getWriters();
    private PrintWriter stats = outFiles.getWriter("stats");
    private PrintWriter avgfit = outFiles.getWriter("avgfit");
    private PrintWriter bestfit = outFiles.getWriter("bestfit");
    private PrintWriter worstfit = outFiles.getWriter("worstfit");
    private PrintWriter size = outFiles.getWriter("size");

    private final String ln = System.getProperty("line.separator");

    private Instance[] instances;   // The set of training instances
    private RuleSet[] ruleSets; // The GA population

    private RuleSet bestRuleSet;  // Best rule set found in the current GA iteration
    private RuleSet bestEverRuleSet;  // Best ever rule set found

    private int generation;  // Number of the current generation
    private boolean foundPerfect = false;  // Has a perfect rule set been found?
    private int foundPerfectAt;  // Generation at which a perfect rule set was found


    /***** Read in the training instances and initialise the GA population *****/

    public Population() {

        if (instantiated) {
            System.out.println("Error: population already instantiated\n");
            System.exit(1);
        }

        instantiated = true;
        getInstances();
        initialiseGA();

    } //end Population()


    /***** Read the instances from a text file into an array *****/

    public void getInstances() {

        instances = new Instance[params.getNumInstances()];

        System.out.println("\nRetrieving instances...\n");

        BufferedReader in = null;

        // Attempt to get the instance file for reading
        try {
            in = new BufferedReader(new FileReader("instances.txt"));
        }
        catch(FileNotFoundException e) {
            System.out.println("File not found error: " + e + "\n");
            System.exit(1);
        }

        int i;
        String nextInstance = null;


        for (i = 0; i < params.getNumInstances(); i++) {

            // Read in the next instance
            try {
                nextInstance = in.readLine();
```

```
        }
        catch(IOException e) {
            System.out.println("Input/output error: " + e + "\n");
            System.exit(1);
        }

        // Add instance to the array
        if (nextInstance == null) {
            System.out.println("Error: not enough instances\n");
            System.exit(1);
        }
        instances[i] = new Instance(nextInstance);

    }

    // Shuffle the instances randomly
    int rand;
    Instance temp;
    for (i = 0; i < params.getNumInstances(); i++) {
        rand = generator.nextInt(params.getNumInstances());
        temp = instances[i];
        instances[i] = instances[rand];
        instances[rand] = temp;
    }

} //end getInstances()


/***** Initialise the population for the GA *****/

public void initialiseGA() {

    ruleSets = new RuleSet[params.getPopSize()];

    System.out.println("Initialising GA population...\n");

    // Initialise the individuals with random bits and evaluate them
    for (int i = 0; i < params.getPopSize(); i++) {
        ruleSets[i] = new RuleSet(params.getInitialNumRules());
        ruleSets[i].initialise();
        ruleSets[i].evaluate(instances);
    }

    // Initialise best rule set and best ever rule set
    bestRuleSet = new RuleSet(0);
    bestEverRuleSet = new RuleSet(0);
    bestEverRuleSet.setFitness(-1);

    // Initialise the generation counter
    generation = 1;

    // Output the initial population
    outputStats();

} //end initialiseGA()
```

```
/***** Find and output the best rule set and current fitness statistics *****/

public void outputStats() {

    System.out.println("Outputting stats for generation " + generation + "...");

    int i;

    // Find the best rule set in the current generation
    bestRuleSet.setFitness(-1);
    for (i = 0; i < params.getPopSize(); i++) {
        if (ruleSets[i].getFitness() > bestRuleSet.getFitness()
        || (ruleSets[i].getFitness() == bestRuleSet.getFitness()
        && ruleSets[i].getNumMacroClassifiers() < bestRuleSet.getNumMacroClassifiers())) {
            bestRuleSet.copyOf(ruleSets[i]);
        }
    }

    // Update best ever rule set if appropriate
    if (bestRuleSet.getFitness() > bestEverRuleSet.getFitness()
    || (bestRuleSet.getFitness() == bestEverRuleSet.getFitness()
    && bestRuleSet.getNumMacroClassifiers() < bestEverRuleSet.getNumMacroClassifiers())) {
        bestEverRuleSet.copyOf(bestRuleSet);
    }

    // Make a note of when a perfect rule set first appears
    if (bestEverRuleSet.getFitness() == 1.0 && !foundPerfect) {
        foundPerfect = true;
        foundPerfectAt = generation;
    }

    // Output the number of macroclassifiers in the best ever rule set
    size.println(generation + ", " + bestEverRuleSet.getNumMacroClassifiers());

    // Calculate the best, worst and average fitnesses for this generation
    double averageFitness = ruleSets[0].getFitness();
    double bestFitness = ruleSets[0].getFitness();
    double worstFitness = ruleSets[0].getFitness();
    for (i = 1; i < params.getPopSize(); i++) {
        averageFitness += ruleSets[i].getFitness();
        if (ruleSets[i].getFitness() > bestFitness)
            bestFitness = ruleSets[i].getFitness();
        if (ruleSets[i].getFitness() < worstFitness)
            worstFitness = ruleSets[i].getFitness();
    }
    averageFitness /= params.getPopSize();
    avgfit.println(generation + ", " + averageFitness);
    bestfit.println(generation + ", " + bestFitness);
    worstfit.println(generation + ", " + worstFitness);

    // Output the current population of rule sets and the best ones found
    if (generation == 1
    || generation == params.getMaxGenerations()
    || generation % params.getOutputFrequency() == 0) {

        stats.println("-------------------------------------------------" + ln);
        stats.println("GENERATION " + generation + ln);
        stats.println("-------------------------------------------------" + ln);
```

```
        for (i = 0; i < params.getPopSize(); i++) {
            stats.println("RULE SET #" + i);
            ruleSets[i].printSortedToFile(stats);
            stats.println();
        }
        stats.println("-------------------------------------------------" + ln);
        stats.println("Best rule set in generation " + generation + ":" + ln);
        bestRuleSet.printSortedToFile(stats);
        stats.println(ln + "Best ever rule set:" + ln);
        bestEverRuleSet.printSortedToFile(stats);
        stats.println(ln + "Average fitness in this generation: "
                      + averageFitness + ln);
    }

} //end outputStats()



/***** Create the next generation *****/

public void runGA() {

    int i, j;

    // Increment the generation counter
    generation++;

    // Select parents for the next generation
    ruleSets = genops.tournamentSelection(ruleSets);

    // Cross each pair of parents according to the crossover probability
    for (i = 0; i < params.getPopSize() - 1; i += 2) {
        if (generator.nextDouble() < params.getCrossoverRate()) {
            genops.twoPtCrossover(ruleSets[i], ruleSets[i+1]);
            ruleSets[i].evaluate(instances);
            ruleSets[i+1].evaluate(instances);
        }
    }

    // Mutate bits in each rule set according to the mutation probability
    for (i = 0; i < params.getPopSize(); i++) {
        for (j = 0; j < ruleSets[i].getLength(); j++) {
            if (generator.nextDouble() < params.getMutationRate()) {
                genops.mutate(ruleSets[i], j);
                ruleSets[i].evaluate(instances);
            }
        }
    }

} //end runGA()



/***** Perform the breed / evaluate cycle for as long as necessary
       or specified *****/

public void adapt() {

    System.out.println("Breed / evaluate cycle started...\n");

        while (generation < params.getMaxGenerations()) {
            runGA();
            outputStats();
        }

        System.out.println("GA terminated after " + params.getMaxGenerations() +" generations.");

        if (foundPerfect) {
            System.out.println("Perfect rule set found after "+foundPerfectAt+" generations.\n");
            stats.println("-------------------------------------------------" + ln);
            stats.println("Perfect rule set found after "+ foundPerfectAt + " generations." + ln);
        }
        else {
            System.out.println("No perfect rule set was found.\n");
            stats.println("-------------------------------------------------" + ln);
            stats.println("No perfect rule set was found." + ln);
        }

        System.out.println("Best ever rule set:\n");
        bestEverRuleSet.printSorted();
        System.out.print("\n");

    } //end adapt()


} //end class Population
```

RuleSet

```
import java.util.*;

/* An individual set of classifiers. */

public class RuleSet {

    private int numRules;              // Number of classifiers in the rule set
    private Vector classifiers;        // The set of classifiers
    private Vector macroClassifiers;   // The set of classifiers without duplication
    private int totalLength;      // Length of the string of concatenated classifiers
    private double fitness;            // Current fitness value of the rule set

    private Parameters params = Parameters.getParams();
    private RandomNumbers generator = RandomNumbers.getGenerator();



    /***** Constructor method *****/

    public RuleSet(int numberOfRules) {

        if (numberOfRules < 0) {
            System.out.println("Error: invalid size specified for rule set\n");
            System.exit(1);
        }
```

```
    numRules = numberOfRules;
    classifiers = new Vector();
    totalLength = numRules * params.getRuleLength();
    fitness = 0.0;

} //end RuleSet()


/***** Initialise with random classifiers *****/

public void initialise() {

    for (int i = 0; i < numRules; i++) { classifiers.add(new Classifier()); }
    createMacroClassifiers();

} //end initialise()



/***** Create a list of the classifiers without duplicates *****/

public void createMacroClassifiers() {

    macroClassifiers = new Vector();
    Classifier thisClassifier;

    int i, j;
    boolean listed;

    // Loop through each classifier in the rule set
    for (i = 0; i < numRules; i++) {

        thisClassifier = (Classifier)classifiers.get(i);

        j = 0;
        listed = false;

        // Is this classifier already listed among the macroclassifiers?
        while (j < macroClassifiers.size() && listed == false) {
            if (((MacroClassifier)macroClassifiers.get(j)).isEqualTo(thisClassifier))
                listed = true;
            j++;
        }

        // If it is, just increment that macroclassifier's numerosity value
        if (listed) {
            ((MacroClassifier)macroClassifiers.get(j-1)).incrementNumerosity();
            thisClassifier.setMacroIndex(j-1);
        }

        // If not, add it to the set of macroclassifiers
        else {
            macroClassifiers.add(new MacroClassifier((Classifier)classifiers.get(i)));
            thisClassifier.setMacroIndex(j);
        }

    }

} //end createMacroClassifiers()
```

```
/***** Print the sorted rule set without duplicated classifiers *****/

public void printSorted() {

    int i;

    Vector sortedMacroClassifiers = new Vector();
    for (i = 0; i < macroClassifiers.size(); i++) {
        sortedMacroClassifiers.add(((MacroClassifier)macroClassifiers.get(i)).toString());
    }
    Collections.sort(sortedMacroClassifiers);

    System.out.println(macroClassifiers.size() + " macroclassifiers:");
    for (i = 0; i < macroClassifiers.size(); i++) {
        System.out.println((String)sortedMacroClassifiers.get(i));
    }
    System.out.println("Total length: " + totalLength + " (" + numRules + " rules)");
    System.out.println("Fitness: " + fitness);

} //end printSorted()


/***** Print the sorted rule set without duplicated classifiers to the specified
       file *****/

public void printSortedToFile(PrintWriter file) {

    int i;

    Vector sortedMacroClassifiers = new Vector();
    for (i = 0; i < macroClassifiers.size(); i++) {
        sortedMacroClassifiers.add(((MacroClassifier)macroClassifiers.get(i)).toString());
    }
    Collections.sort(sortedMacroClassifiers);

    file.println(macroClassifiers.size() + " macroclassifiers:");
    for (i = 0; i < macroClassifiers.size(); i++) {
        file.println((String)sortedMacroClassifiers.get(i));
    }
    file.println("Total length: " + totalLength + " (" + numRules + " rules)");
    file.println("Fitness: " + fitness);

} //end printSortedToFile()



/***** Return the number of classifiers in the rule set *****/

public int getNumRules() {

    return numRules;

} //end getNumRules()



/***** Return the number of macroclassifiers in the rule set *****/

public int getNumMacroClassifiers() {
```

```
    return macroClassifiers.size();

} //end getNumMacroClassifiers()


/***** Return the total length of the rule set in bits *****/

public int getLength() {

    return totalLength;

} //end getLength()


/***** Return the current fitness of the rule set *****/

public double getFitness() {

    return fitness;

} //end getFitness()


/***** Assign a fitness value to the rule set *****/

public void setFitness(double fitVal) {

    fitness = fitVal;

} //end setFitness()


/***** Return a concatenation of the classifiers *****/

public String getRules() {

    StringBuffer buffer = new StringBuffer();
    for (int i = 0; i < numRules; i++) {
        buffer.append(((Classifier)classifiers.get(i)).getCondition());
        buffer.append(((Classifier)classifiers.get(i)).getAction());
    }
    return buffer.toString();

} //end getRules()


/***** Replace the entire set of classifiers *****/

public void setRules(String newRules) {

    if (newRules.length() % params.getRuleLength() != 0) {
        System.out.println("Error: invalid classifier string supplied to rule set\n");
        System.exit(1);
    }

    // Remove any existing classifiers from the rule set
    classifiers.clear();

    // Update length and classifier count variables
    totalLength = newRules.length();
    numRules = totalLength / params.getRuleLength();

    // Create new classifiers and add them to this rule set
    String condition, action;
    for (int i = 0; i < totalLength; i += params.getRuleLength()) {
        condition = newRules.substring(i, i + params.getCondLength());
        action = newRules.substring(i + params.getCondLength(), i + params.getRuleLength());
        classifiers.add(new Classifier(condition, action));
    }

    // Update the rule set's macroclassifiers
    createMacroClassifiers();

    // Reset the fitness as the rule set will need re-evaluating
    fitness = 0.0;

} //end setRules()


/***** Make this rule set into a copy of another one *****/

public void copyOf(RuleSet other) {

    // Copy the classifiers to this rule set and update appropriate variables
    setRules(other.getRules());

    // Make fitness equal to that of the other rule set
    fitness = other.getFitness();

} //end copyOf()


/***** Calculate the rule set's fitness value and delete any unwanted
       classifiers *****/

public void evaluate(Instance[] instances) {

    double rawFitness = 0.0;
    double meanActionSetSize = 0.0;
    int i, j;
    MacroClassifier thisClassifier;

    // Compute accuracy of each macroclassifier in the rule set
    for (i = 0; i < macroClassifiers.size(); i++) {
        ((MacroClassifier)macroClassifiers.get(i)).evaluate(instances);
        ((MacroClassifier)macroClassifiers.get(i)).resetActionSetStats();
    }

    String message, classification;
    Vector actionSet = new Vector();
    int copies, actionSetSize;
```

```
double actionSetAccuracy;
MacroClassifier maxGenClassifier;
int highestGenerality, index;
boolean ruleDeletionActivated = false;

// Loop through all data instances
for (i = 0; i < params.getNumInstances(); i++) {

    message = instances[i].getMessage();
    classification = instances[i].getClassification();
    actionSet.clear();
    actionSetSize = 0;
    actionSetAccuracy = 0.0;

    // Add to action set any matched macroclassifiers that correctly classify
    // the instance
    for (j = 0; j < macroClassifiers.size(); j++) {
        thisClassifier = (MacroClassifier)macroClassifiers.get(j);
        if (!thisClassifier.isToBeSubsumed()
        && thisClassifier.isMatchedBy(message)
        && (thisClassifier.getAction()).equals(classification)) {
            actionSet.add((MacroClassifier)macroClassifiers.get(j));
            copies = thisClassifier.getNumerosity();
            actionSetSize += copies;
            actionSetAccuracy += thisClassifier.getAccuracy() * copies;
        }
    }
    if (actionSetSize > 0) actionSetAccuracy /= actionSetSize;

    // Update stats for each classifier in the action set
    for (j = 0; j < actionSet.size(); j++) {
        thisClassifier = (MacroClassifier)actionSet.get(j);
        thisClassifier.addToMeanActionSetSize(actionSetSize);
        thisClassifier.addToMeanActionSetAccuracy(actionSetAccuracy);
    }

    // Update stats for the rule set
    meanActionSetSize += actionSetSize;
    rawFitness += actionSetAccuracy;

    if (actionSet.size() > 1) {

        // Find the index of the most general accurate classifier in the
        // action set
        highestGenerality = 0;
        index = -1;
        for (j = 0; j < actionSet.size(); j++) {
            thisClassifier = (MacroClassifier)actionSet.get(j);
            if (thisClassifier.getAccuracy() == 1.0
            && thisClassifier.getNumActionSets() > 1
            && thisClassifier.getGenerality() > highestGenerality) {
                highestGenerality = thisClassifier.getGenerality();
                index = j;
            }
        }

        // If such a classifier was found, accurate subsumed classifiers can
        // be replaced
        if (index > -1) {
            maxGenClassifier = (MacroClassifier)actionSet.get(index);

            // Loop through the rest of the action set, marking classifiers
            // for subsumption
            for (j = 0; j < index; j++) {
                thisClassifier = (MacroClassifier)actionSet.get(j);
                if (thisClassifier.getAccuracy() == 1.0
                && maxGenClassifier.subsumes(thisClassifier)
                && generator.nextDouble() < params.getSubsumptionProbability()) {
                    thisClassifier.markForSubsumption(maxGenClassifier.getCondition(),
                                                      maxGenClassifier.getAction());
                    ruleDeletionActivated = true;
                    maxGenClassifier.addToNumerosity(thisClassifier.getNumerosity());
                }
            }
            for (j = index + 1; j < actionSet.size(); j++) {
                thisClassifier = (MacroClassifier)actionSet.get(j);
                if (thisClassifier.getAccuracy() == 1.0
                && maxGenClassifier.subsumes(thisClassifier)
                && generator.nextDouble() < params.getSubsumptionProbability()) {
                    thisClassifier.markForSubsumption(maxGenClassifier.getCondition(),
                                                      maxGenClassifier.getAction());
                    ruleDeletionActivated = true;
                    maxGenClassifier.addToNumerosity(thisClassifier.getNumerosity());
                }
            }

        } //end if

    } //end if

} //end for

// Calculate the rule set's overall fitness value
fitness = rawFitness / params.getNumInstances();

// Calculate the average size of the action sets
meanActionSetSize /= params.getNumInstances();

// Loop through the macroclassifiers, marking the worst for deletion
for (i = 0; i < macroClassifiers.size(); i++) {

    thisClassifier = (MacroClassifier)macroClassifiers.get(i);

    // Skip over classifiers already marked for subsumption deletion
    if (thisClassifier.isToBeSubsumed()) { }

    // Delete any classifiers that are 100% inaccurate
    else if (thisClassifier.getNumActionSets() == 0) {
        thisClassifier.markForDeletion();
        ruleDeletionActivated = true;
    }

    else {
        thisClassifier.computeMeanActionSetSize();
        thisClassifier.computeMeanActionSetAccuracy();

        // Delete below-average classifiers that appear in large action sets
        if (thisClassifier.getAccuracy() < thisClassifier.getMeanActionSetAccuracy()) {
```

```
                if (generator.nextDouble() < params.getDeletionProbability()
                && thisClassifier.getMeanActionSetSize() >= meanActionSetSize) {
                    thisClassifier.markForDeletion();
                    ruleDeletionActivated = true;
                }
            }
        }

    } //end for

    // Loop through the rule set, removing classifiers marked for deletion
    if (ruleDeletionActivated) {

        int macroIndex;
        MacroClassifier theMacroClassifier;

        i = 0;
        while (i < numRules && numRules > 1) {

            macroIndex = ((Classifier)classifiers.get(i)).getMacroIndex();
            theMacroClassifier = (MacroClassifier)macroClassifiers.get(macroIndex);

            if (theMacroClassifier.isToBeDeleted()) {
                classifiers.remove(i);
                numRules--;
                totalLength -= params.getRuleLength();
            }

            else if (theMacroClassifier.isToBeSubsumed()) {
                classifiers.set(i, new Classifier(theMacroClassifier.getCondition(),
                                                  theMacroClassifier.getAction()));
                i++;
            }

            else i++;

        }

        createMacroClassifiers();   // Update the rule set's macroclassifiers

    }

  } //end evaluate()


} //end class RuleSet
```

Classifier

```
/* A single classifier within a rule set. */

public class Classifier {

    private String condition;   // Condition string for messages to be matched against
    private String action;      // Prediction as to the class of a matched instance
    private int generality;     // Number of wildcards in the condition string
    private int macroIndex;     // Index of the corresponding macroclassifier

    private Parameters params = Parameters.getParams();
    private RandomNumbers generator = RandomNumbers.getGenerator();


    /***** Create a randomised classifier *****/

    public Classifier() {

        StringBuffer buffer = new StringBuffer();
        String nextBit;
        int i;

        generality = 0;

        for (i = 0; i < params.getCondLength(); i++) {
            if (Math.random() < params.getGenerality()) {
                nextBit = "#";
                generality++;
            }
            else {
                nextBit = Integer.toString(generator.nextInt(2));
            }
            buffer.append(nextBit);
        }

        condition = buffer.toString();

        buffer.setLength(0);

        for (i = 0; i < params.getActLength(); i++) {
            nextBit = Integer.toString(generator.nextInt(2));
            buffer.append(nextBit);
        }

        action = buffer.toString();

    } //end Classifier()


    /***** Create a classifier with the given condition and action strings *****/

    public Classifier(String cond, String act) {

        if (cond.length() != params.getCondLength()
        || act.length() != params.getActLength()) {
            System.out.println("Error: invalid string supplied to new classifier method\n");
            System.exit(1);
        }
        condition = cond;
        action = act;

        generality = 0;
        for (int i = 0; i < params.getCondLength(); i++) {
            if (condition.charAt(i) == '#') generality++;
        }

    } //end Classifier()
```

```
    /***** Return the condition string *****/

    public String getCondition() {

        return condition;

    } //end getCondition()


    /***** Return the action string *****/

    public String getAction() {

        return action;

    } //end getAction()


    /***** Return the number of wildcards in the classifier's condition string *****/

    public int getGenerality() {

        return generality;

    } //end getGenerality()


    /***** Record the index of the corresponding macroclassifier *****/

    public void setMacroIndex(int index) {

        macroIndex = index;

    } //end setMacroIndex()


    /***** Return the index of the corresponding macroclassifier *****/

    public int getMacroIndex() {

        return macroIndex;

    } //end getMacroIndex()


} //end class Classifier
```

MacroClassifier

```
/* Counts the number of duplicates of a particular classifier. Accuracy evaluation is
   done on the macroclassifiers to avoid unnecessary processing. */

public class MacroClassifier {

    private String condition;   // Condition string for messages to be matched against
    private String action;      // Prediction as to the class of a matched instance
    private int generality;     // Number of wildcards in the condition string
    private int numerosity;     // Number of copies of the classifier in the rule set

    private double accuracy;  // Correct predictions over number of matching instances
    private double meanActionSetSize;  // Mean size of action sets the rule appears in
    private double meanActionSetAccuracy;  // Mean accuracy of action sets appeared in
    private int numActionSets;  // Number of action sets the classifier appears in
    private boolean toBeDeleted = false;  // Is the classifier marked for deletion?
    private boolean toBeSubsumed = false; // Is the classifier marked for subsumption?

    private Parameters params = Parameters.getParams();


    /***** Constructor method *****/

    public MacroClassifier(Classifier microclassifier) {

        condition = microclassifier.getCondition();
        action = microclassifier.getAction();
        generality = microclassifier.getGenerality();
        numerosity = 1;

    } //end MacroClassifier()


    /***** Increment counter when another copy of the classifier is found *****/

    public void incrementNumerosity() {

        numerosity++;

    } //end incrementNumerosity()


    /***** Add the given amount to the numerosity *****/

    public void addToNumerosity(int amount) {

        numerosity += amount;

    } //end addToNumerosity()


    /***** Return the classifier's condition string *****/

    public String getCondition() {

        return condition;

    } //end getCondition()
```

```
/***** Return the classifier's action string *****/

public String getAction() {

    return action;

} //end getAction()


/***** Return the classifier's generality *****/

public int getGenerality() {

    return generality;

} //end getGenerality()


/***** Return the number of copies of the classifier *****/

public int getNumerosity() {

    return numerosity;

} //end getNumerosity()


/***** Create a string representation of the macroclassifier *****/

public String toString() {

    return condition + ":" + action + " (" + numerosity + ")";

} //end toString()


/***** Return true if the given rule is a duplicate of this macroclassifier *****/

public boolean isEqualTo(Classifier microclassifier) {

    if (condition.equals(microclassifier.getCondition())
    && action.equals(microclassifier.getAction()))
        return true;
    else
        return false;

} //end isEqualTo()


/***** Return true if the classifier is matched by the given message *****/

public boolean isMatchedBy(String message) {

    if (message.length() != condition.length()) {
        System.out.println("Error: invalid message supplied to match method\n");
        System.exit(1);
    }

    int i = 0;
    boolean matched = true;

    while (i < condition.length() && matched == true) {
        if (message.charAt(i) != condition.charAt(i)
        && condition.charAt(i) != '#')
            matched = false;
        i++;
    }

    return matched;

} //end isMatchedBy()


/***** Calculate the classifier's accuracy *****/

public void evaluate(Instance[] instances) {

    int numMatches = 0;
    int numCorrect = 0;

    for (int i = 0; i < params.getNumInstances(); i++) {
        if (this.isMatchedBy(instances[i].getMessage())) {
            numMatches++;
            if (action.equals(instances[i].getClassification())) numCorrect++;
        }
    }

    if (numMatches == 0) accuracy = 0.0;
    else accuracy = numCorrect / (double)numMatches;

    numActionSets = numCorrect;

} //end evaluate()


/***** Return the accuracy of the classifier *****/

public double getAccuracy() {

    return accuracy;

} //end getAccuracy()


/***** Return true if this classifier subsumes the given classifier *****/

public boolean subsumes(MacroClassifier other) {

    String thisRule = condition + action;
    String otherRule = other.getCondition() + other.getAction();
```

```
    int i = 0;
    boolean subsumes = true;

    while (i < params.getRuleLength() && subsumes == true) {
        if (otherRule.charAt(i) != thisRule.charAt(i)
        && thisRule.charAt(i) != '#')
            subsumes = false;
        i++;
    }

    return subsumes;

} //end subsumes()


/***** Set the action set variables to zero *****/

public void resetActionSetStats() {

    meanActionSetSize = 0;
    meanActionSetAccuracy = 0.0;

} //end resetActionSetStats()


/***** Increase meanActionSetSize by the given amount *****/

public void addToMeanActionSetSize(int newSize) {

    meanActionSetSize += newSize;

} //end addToMeanActionSetSize()


/***** Increase meanActionSetAccuracy by the given amount*****/

public void addToMeanActionSetAccuracy(double newAccuracy) {

    meanActionSetAccuracy += newAccuracy;

} //end addToMeanActionSetAccuracy()


/***** Divide cumulative action set size by the number of action sets the rule
       appears in *****/

public void computeMeanActionSetSize() {

    meanActionSetSize /= numActionSets;

} //end computeMeanActionSetSize()
```

```
/***** Divide cumulative action set accuracy by number of action sets the rule
       appears in *****/

public void computeMeanActionSetAccuracy() {

    meanActionSetAccuracy /= numActionSets;

} //end computeMeanActionSetAccuracy()


/***** Return the average size of the action sets the rule appears in *****/

public double getMeanActionSetSize() {

    return meanActionSetSize;

} //end getMeanActionSetSize()


/***** Return the average accuracy of the action sets the rule appears in *****/

public double getMeanActionSetAccuracy() {

    return meanActionSetAccuracy;

} //end getMeanActionSetAccuracy()


/***** Return the number of action sets the rule appears in *****/

public int getNumActionSets() {

    return numActionSets;

} //end getNumActionSets()


/***** Mark the classifier for deletion *****/

public void markForDeletion() {

    toBeDeleted = true;

} //end markForDeletion()


/***** Return true if the classifier has been marked for deletion *****/

public boolean isToBeDeleted() {

    return toBeDeleted;

} //end isToBeDeleted()
```

```
    /***** Mark the classifier for subsumption and keep the replacement strings *****/

    public void markForSubsumption(String newCond, String newAct) {

        toBeSubsumed = true;
        condition = newCond;
        action = newAct;

    } //end markForSubsumption()


    /***** Return true if the classifier has been marked for subsumption *****/

    public boolean isToBeSubsumed() {

        return toBeSubsumed;

    } //end isToBeSubsumed()

} // end class MacroClassifier
```

Instance

```
/* An example and the class it belongs to. */

public class Instance {

    private String message;         // Message to match against classifiers
    private String classification;  // The class that the instance belongs to

    private Parameters params = Parameters.getParams();


    /***** Constructor method *****/

    public Instance(String bits) {

        if (bits.length() != params.getRuleLength()) {
            System.out.println("Error: invalid string supplied to new instance method\n");
            System.exit(1);
        }
        message = bits.substring(0, params.getCondLength());
        classification = bits.substring(params.getCondLength());

    } //end Instance()


    /***** Return the message part of the instance *****/

    public String getMessage() {

        return message;

    } //end getMessage()


    /***** Return the class of the instance *****/

    public String getClassification() {

        return classification;

    } //end getClassification()


} //end class Instance
```

GeneticOps

```
/* Operators for selection, crossover and mutation of rule sets. */

public class GeneticOps {

    private Parameters params = Parameters.getParams();
    private RandomNumbers generator = RandomNumbers.getGenerator();


    /***** Constructor method *****/

    public GeneticOps() { }


    /***** Tournament selection *****/

    public RuleSet[] tournamentSelection(RuleSet[] ruleSets) {

        RuleSet[] newPopulation = new RuleSet[params.getPopSize()];
        int i, j;
        int[] selectionPool = new int[params.getTournamentSize()];
        int c1, c2, better, selected;
        double sumFitness, rand, cumulative;

        // Repeat until a whole new population is created
        for (i = 0; i < params.getPopSize(); i++) {

            // Initialise the new rule set
            newPopulation[i] = new RuleSet(params.getInitialNumRules());

            sumFitness = 0.0;

            // Create a selection pool
            for (j = 0; j < params.getTournamentSize(); j++) {

                // Select two rule sets from the existing population at random
                c1 = generator.nextInt(params.getPopSize());
                c2 = generator.nextInt(params.getPopSize());

                // Determine the 'better' of the two candidates
                if (ruleSets[c1].getFitness() > ruleSets[c2].getFitness()) {

                    // If fitness difference < threshold, shorter candidate is better
```

```
                if (ruleSets[c1].getFitness() - ruleSets[c2].getFitness()
                < params.getFitnessThreshold()) {
                   better = (ruleSets[c1].getLength() <= ruleSets[c2].getLength()) ? c1 : c2;
                }

                // Otherwise take the fitter candidate
                else better = c1;
            }
            else if (ruleSets[c2].getFitness() > ruleSets[c1].getFitness()) {

                // If fitness difference < threshold, shorter candidate is better
                if (ruleSets[c2].getFitness() - ruleSets[c1].getFitness()
                < params.getFitnessThreshold()) {
                   better = (ruleSets[c2].getLength() <= ruleSets[c1].getLength()) ? c2 : c1;
                }

                // Otherwise take the fitter candidate
                else better = c2;
            }
            else {
                // If fitnesses are equal, take the shorter candidate
                better = (ruleSets[c1].getLength() <= ruleSets[c2].getLength()) ? c1 : c2;
            }

            // Add the successful candidate's index to the selection pool
            selectionPool[j] = better;
            sumFitness += ruleSets[better].getFitness();

        }

        // Choose an individual from the pool using roulette wheel selection
        rand = sumFitness * generator.nextDouble();
        j = 0;
        cumulative = 0.0;
        while (cumulative <= rand) {
            cumulative += ruleSets[selectionPool[j]].getFitness();
            j++;
        }
        selected = selectionPool[j-1];

        // Copy the chosen rule set to the new population
        newPopulation[i].copyOf(ruleSets[selected]);

    } //end for

    return newPopulation;

} //end tournamentSelection()


/***** Two point crossover *****/

public void twoPtCrossover(RuleSet par1, RuleSet par2) {

    int r1, r2, o1, o2;
    int par1rule1, par1rule2, par2rule1, par2rule2, offset1, offset2;

    // Pick two random classifiers in parent 1
    r1 = generator.nextInt(par1.getNumRules());
    r2 = generator.nextInt(par1.getNumRules());
    if (r1 < r2) { par1rule1 = r1; par1rule2 = r2; }
    else { par1rule1 = r2; par1rule2 = r1; }

    // Pick two random classifiers in parent 2
    r1 = generator.nextInt(par2.getNumRules());
    r2 = generator.nextInt(par2.getNumRules());
    if (r1 < r2) { par2rule1 = r1; par2rule2 = r2; }
    else { par2rule1 = r2; par2rule2 = r1; }

    // Choose two random offsets from a rule boundary
    o1 = generator.nextInt(params.getRuleLength());
    o2 = generator.nextInt(params.getRuleLength());
    offset1 = o1;
    offset2 = o2;

    // If both cross points are within the same rule, check they are in order
    if (par1rule1 == par1rule2 || par2rule1 == par2rule2) {
        if (o1 > o2) {
            offset1 = o2;
            offset2 = o1;
        }
    }

    // Work out the crossover positions
    int crossPt1_1 = par1rule1 * params.getRuleLength() + offset1;
    int crossPt1_2 = par1rule2 * params.getRuleLength() + offset2;
    int crossPt2_1 = par2rule1 * params.getRuleLength() + offset1;
    int crossPt2_2 = par2rule2 * params.getRuleLength() + offset2;

    // Perform crossover on the strings of classifiers
    String parent1 = par1.getRules();
    String parent2 = par2.getRules();
    String child1 = parent1.substring(0, crossPt1_1) +
                    parent2.substring(crossPt2_1, crossPt2_2) +
                    parent1.substring(crossPt1_2);

    String child2 = parent2.substring(0, crossPt2_1) +
                    parent1.substring(crossPt1_1, crossPt1_2) +
                    parent2.substring(crossPt2_2);

    // Replace the parents with their offspring
    par1.setRules(child1);
    par2.setRules(child2);

} //end twoPtCrossover()


/***** Mutation *****/

public void mutate(RuleSet ruleSet, int position) {

    if (position < 0 || position >= ruleSet.getLength()) {
        System.out.println("Error: invalid mutation position supplied\n");
        System.exit(1);
    }
```

```
        String oldRules = ruleSet.getRules();
        char oldValue, newValue;

        // Specified bit is within an action string - just flip the bit
        if (position % params.getRuleLength() >= params.getCondLength()) {
            if (oldRules.charAt(position) == '0') newValue = '1'; else newValue = '0';
        }

        // Specified bit is within a condition string - randomly choose a
        // different value
        else {
            oldValue = oldRules.charAt(position);
            if (oldValue == '0') {
                if (generator.nextDouble() < params.getGenerality()) newValue = '#';
                else newValue = '1';
            }
            else if (oldValue == '1') {
                if (generator.nextDouble() < params.getGenerality()) newValue = '#';
                else newValue = '0';
            }
            else {
                if (generator.nextInt(2) == 0) newValue = '0';
                else newValue = '1';
            }
        }

        // Replace the specified bit with its new value
        String newRules = oldRules.substring(0, position) + newValue +
                          oldRules.substring(position+1);
        ruleSet.setRules(newRules);

    } //end mutate()


} //end class GeneticOps
```